

Direct Sparse Odometry (DSO)

References:

- 1) [EN] DSO paper - <http://vladlen.info/papers/DSO.pdf>
- 2) [KR] SLAMKR study - [SLAMKR Study Season 1](#)
- 3) [CH] DSO code reading - <https://x007dwd.github.io/2017/02/28/dso-slam/>
- 4) [CH] jingeTU's SLAM blog - <https://www.cnblogs.com/JingeTU/category/954354.html>
- 5) [CH] DSO tracking and optimization - <https://blog.csdn.net/xxxlinttp/article/details/90640350>
- 6) [CH] DSO initialization - <https://blog.csdn.net/xxxlinttp/article/details/89379785>
- 7) [CH] Detailed in DSO - <https://zhuanlan.zhihu.com/p/29177540>
- 8) [CH] DSO photometric calibration - <https://www.cnblogs.com/luyb/p/6077478.html>
- 9) [CH] DSO code with comments - <https://github.com/alalagong/DSO>
- 10) [CH] DSO Null Space - <http://www.lingtong.de/2020/04/24/DSO-Null-Space/>
- 11) [CH] DSO (5) Calculation and Derivation of Null Space - <https://blog.csdn.net/wubaobao1993/article/details/105106301>

Last modified: 2023/11/06

1st: 2020/01/02	6th: 2020/01/26	11th: 2020/03/04	16th: 2023/11/06
2nd: 2020/01/06	7th: 2020/02/01	12th: 2020/03/05	
3rd: 2020/01/18	8th: 2020/02/10	13th: 2020/05/17	
4th: 2020/01/20	9th: 2020/02/23	14th: 2020/05/19	
5th: 2020/01/23	10th: 2020/03/03	15th: 2022/12/22	

pdf 보기 탭에서 한 페이지 씩 보기로 설정하시면 ppt 슬라이드 넘기듯이 보실 수 있습니다

개인적으로 공부하기 위해 작성한 자료입니다

내용 중 틀린 부분이나 빠진 내용이 있다면 gyurse@gmail.com 으로 말씀해주시면 감사하겠습니다 :-)

Contents

1. Initialization

- 1.1. Error Function Formulation
- 1.2. Gauss-Newton Optimization
- 1.3. Jacobian Derivation
- 1.4. Solving The Incremental Equation (Schur Complement)

2. Frames

- 2.1. Pose Tracking
- 2.2. Keyframe Decision

3. Non-Keyframes

- 3.1. Inverse Depth Update

Contents

4. Keyframes

4.1. Inverse Depth Update

4.2. Immature Point Activation

4.3. Sliding Window Optimization

 4.3.1. Error Function Formulation

 4.3.2. Jacobian Derivation of Camera Intrinsics

 4.3.3. First Estimate Jacobian (FEJ)

 4.3.4. Adjoint Transformation

 4.3.5. Marginalization

 4.3.6. Solving The Incremental Equation

4.4. Null Space Effect Elimination

5. Pixel Selection

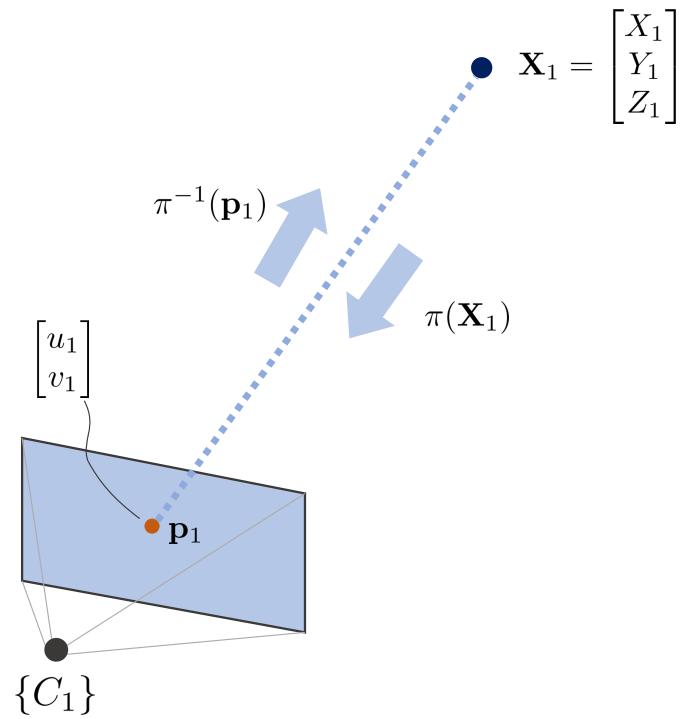
5.1. Make Histogram

5.2. Dynamic Grid

6. DSO Code review

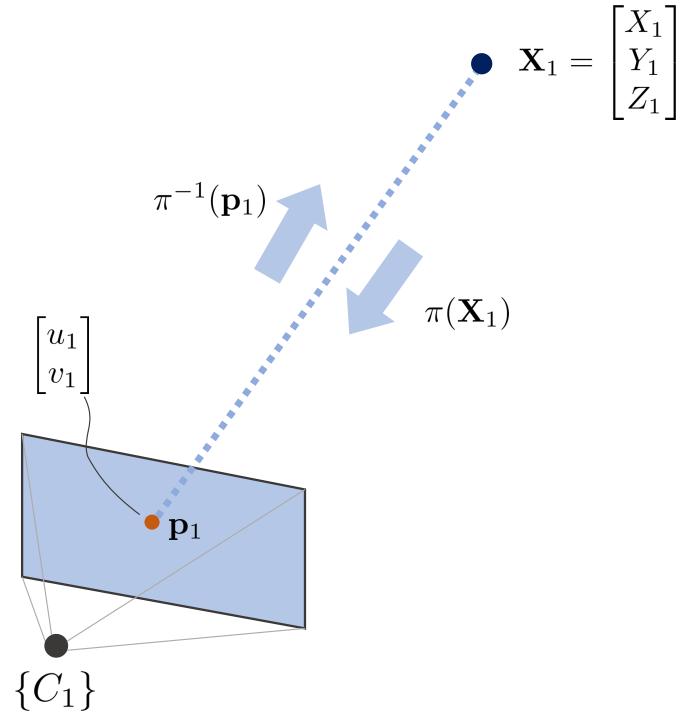
1. Initialization

1.1. Initialization → Error Function Formulation



$\{\mathbf{C}_i\}$	카메라 좌표계
\mathbf{X}_1	3차원 공간 상의 한 점
\mathbf{p}_1	2차원 이미지 평면 상의 한 점
\mathbf{K}	카메라 내부 파라미터(3x3행렬)
ρ_1	깊이 값의 역수 (inverse depth)

1.1. Initialization → Error Function Formulation



3차원 상의 한 점 \mathbf{X}_1 과 2차원 이미지 평면 상의 한 점 \mathbf{p}_1 은 다음과 같은 관계를 가진다.

$$\mathbf{p}_1 = \pi(\mathbf{X}_1) = \mathbf{K}\rho_1\mathbf{X}_1$$

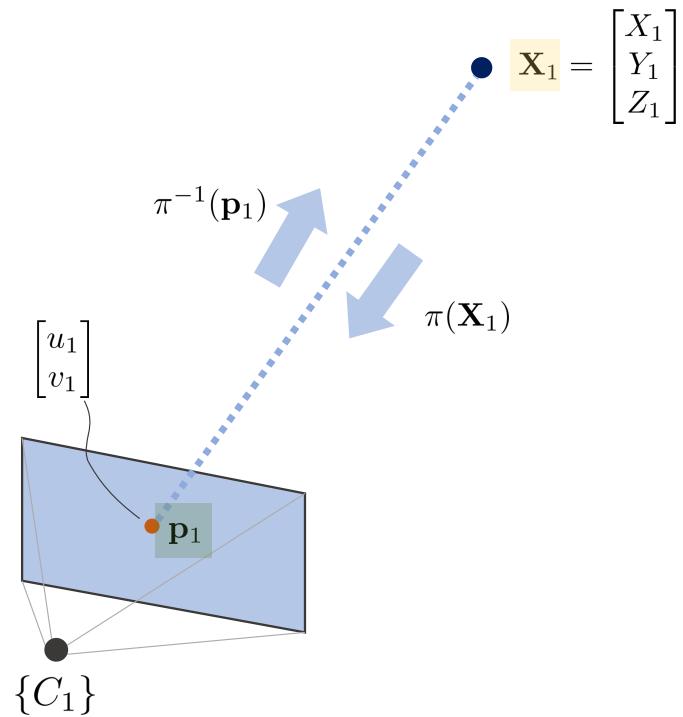
$$\mathbf{X}_1 = \pi^{-1}(\mathbf{p}_1) = \mathbf{K}^{-1}/\rho_1\mathbf{p}_1$$

where, $\rho_1 = 1/Z_1$ (inverse depth)

$$\mathbf{K} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{K}^{-1} = \begin{bmatrix} f_x^{-1} & 0 & -f_x^{-1}c_x \\ 0 & f_y^{-1} & -f_y^{-1}c_y \\ 0 & 0 & 1 \end{bmatrix}$$

$\{\mathbf{C}_i\}$	카메라 좌표계
\mathbf{X}_1	3차원 공간 상의 한 점
\mathbf{p}_1	2차원 이미지 평면 상의 한 점
\mathbf{K}	카메라 내부 파라미터(3x3행렬)
ρ_1	깊이 값의 역수 (inverse depth)

1.1. Initialization → Error Function Formulation



3차원 상의 한 점 \mathbf{X}_1 과 2차원 이미지 평면 상의 한 점 \mathbf{p}_1 은 다음과 같은 관계를 가진다.

$$\boxed{\mathbf{p}_1 = \pi(\mathbf{X}_1) = \mathbf{K}\rho_1 \mathbf{X}_1}$$

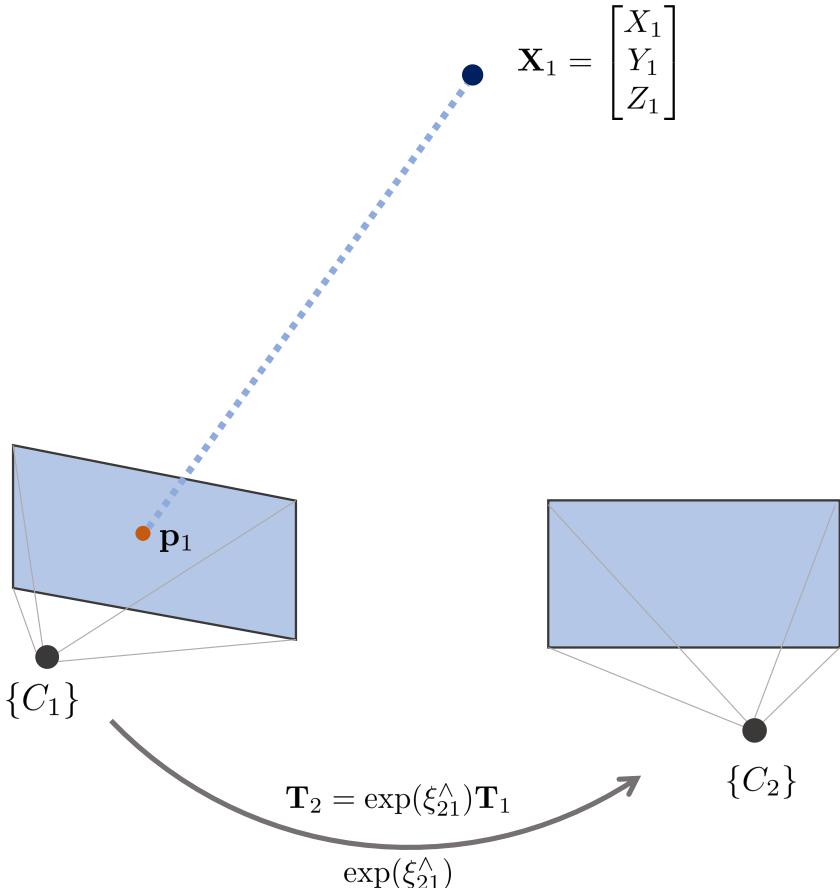
$$\boxed{\mathbf{X}_1 = \pi^{-1}(\mathbf{p}_1) = \mathbf{K}^{-1}/\rho_1 \mathbf{p}_1}$$

where, $\rho_1 = 1/Z_1$ (inverse depth)

$\{\mathbf{C}_i\}$	카메라 좌표계
\mathbf{X}_1	3차원 공간 상의 한 점
\mathbf{p}_1	2차원 이미지 평면 상의 한 점
\mathbf{K}	카메라 내부 파라미터(3x3행렬)
ρ_1	깊이 값의 역수 (inverse depth)

$$\mathbf{K} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{K}^{-1} = \begin{bmatrix} f_x^{-1} & 0 & -f_x^{-1}c_x \\ 0 & f_y^{-1} & -f_y^{-1}c_y \\ 0 & 0 & 1 \end{bmatrix}$$

1.1. Initialization → Error Function Formulation



3차원 상의 한 점 \mathbf{X}_1 과 2차원 이미지 평면 상의 한 점 \mathbf{p}_1 은 다음과 같은 관계를 가진다.

$$\mathbf{p}_1 = \pi(\mathbf{X}_1) = \mathbf{K}\rho_1\mathbf{X}_1$$

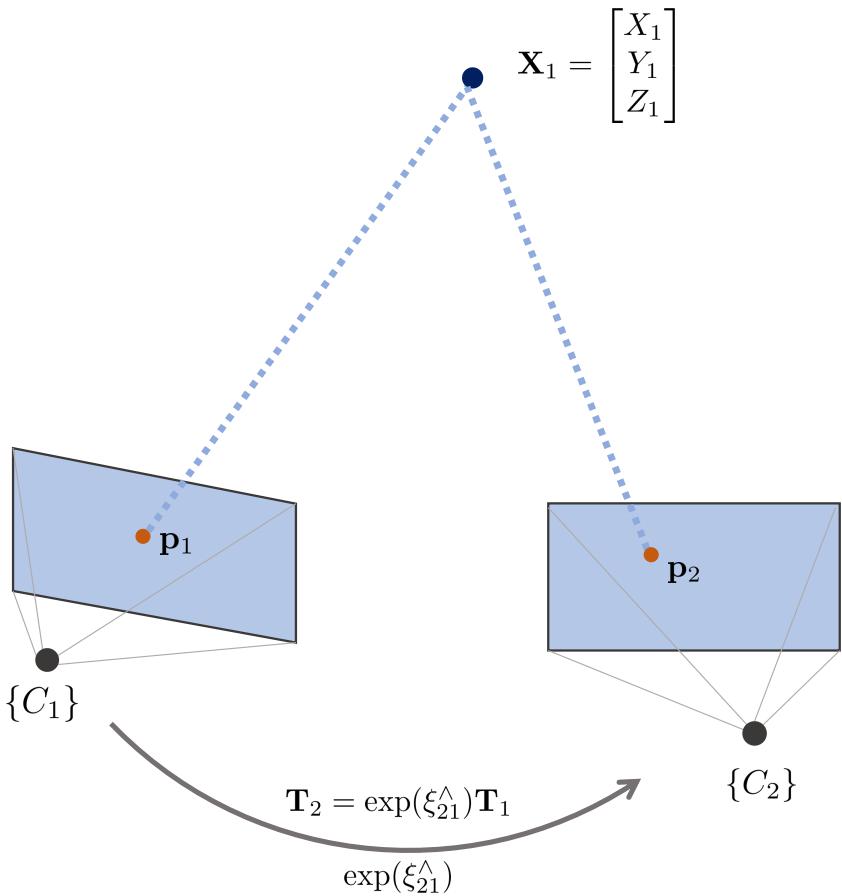
where, $\rho_1 = 1/Z_1$ (inverse depth)

$$\mathbf{X}_1 = \pi^{-1}(\mathbf{p}_1) = \mathbf{K}^{-1}/\rho_1\mathbf{p}_1$$

이 때, 카메라가 $\exp(\xi_{21}^\wedge) \in \text{SE}(3)$ 만큼 움직여서 카메라가 $\{C_1\} \rightarrow \{C_2\}$ 으로 이동한 경우를 생각해보자.

\mathbf{T}_i	카메라의 3차원 포즈 (4x4행렬)
$\xi_{21} \in \mathbb{R}^6$	C1,C2 카메라 간 상대 포즈 변화량 (twist) (6차원벡터)
$\xi_{21}^\wedge \in \text{se}(3)$	hat 연산자가 적용된 twist (lie algebra) (4x4행렬)
$\exp(\xi_{21}^\wedge) \in \text{SE}(3)$	C1,C2 카메라 간 상대 포즈 (transform) (lie group) (4x4행렬)

1.1. Initialization → Error Function Formulation



3차원 상의 한 점 \mathbf{X}_1 과 2차원 이미지 평면 상의 한 점 \mathbf{p}_1 은 다음과 같은 관계를 가진다.

$$\mathbf{p}_1 = \pi(\mathbf{X}_1) = \mathbf{K}\rho_1 \mathbf{X}_1$$

where, $\rho_1 = 1/Z_1$ (inverse depth)

$$\mathbf{X}_1 = \pi^{-1}(\mathbf{p}_1) = \mathbf{K}^{-1}/\rho_1 \mathbf{p}_1$$

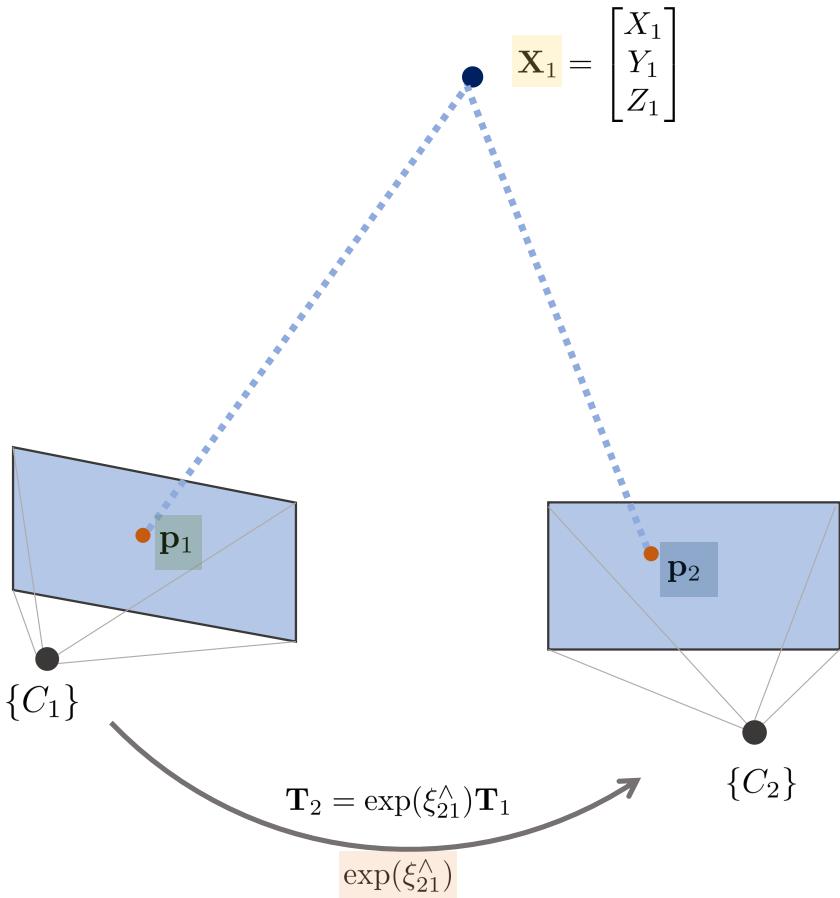
이 때, 카메라가 $\exp(\xi_{21}^\wedge) \in \text{SE}(3)$ 만큼 움직여서 카메라가 $\{C_1\} \rightarrow \{C_2\}$ 으로 이동한 경우를 생각해보자.

\mathbf{X}_1 을 두 번째 카메라 상에 프로젝션한 점 \mathbf{p}_2 는 다음과 같은 공식으로 표현할 수 있다.

$$\mathbf{p}_2 = \pi(\exp(\xi_{21}^\wedge)\mathbf{X}_1) = \pi(\exp(\xi_{21}^\wedge)\pi^{-1}(\mathbf{p}_1))$$

\mathbf{T}_i	카메라의 3차원 포즈 (4x4행렬)
$\xi_{21} \in \mathbb{R}^6$	C1,C2 카메라 간 상대 포즈 변화량 (twist) (6차원벡터)
$\xi_{21}^\wedge \in \text{se}(3)$	hat 연산자가 적용된 twist (lie algebra) (4x4행렬)
$\exp(\xi_{21}^\wedge) \in \text{SE}(3)$	C1,C2 카메라 간 상대 포즈 (transform) (lie group) (4x4행렬)

1.1. Initialization → Error Function Formulation



3차원 상의 한 점 \mathbf{X}_1 과 2차원 이미지 평면 상의 한 점 \mathbf{p}_1 은 다음과 같은 관계를 가진다.

$$\mathbf{p}_1 = \pi(\mathbf{X}_1) = \mathbf{K}\rho_1\mathbf{X}_1$$

where, $\rho_1 = 1/Z_1$ (inverse depth)

$$\mathbf{X}_1 = \pi^{-1}(\mathbf{p}_1) = \mathbf{K}^{-1}/\rho_1\mathbf{p}_1$$

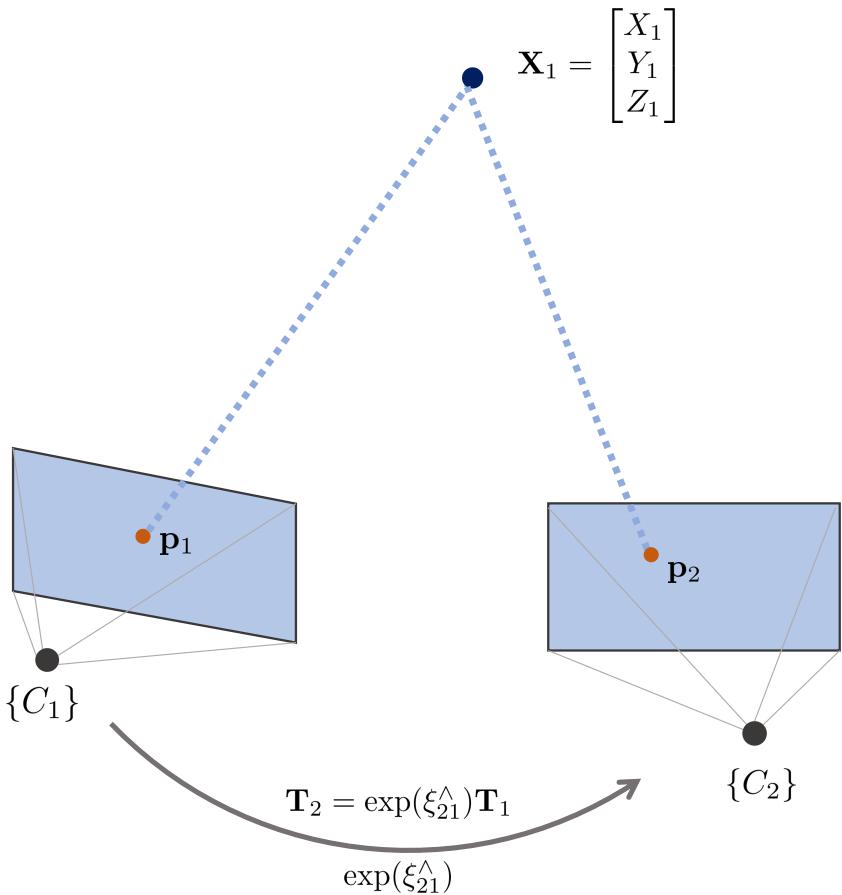
이 때, 카메라가 $\exp(\xi_{21}^\wedge) \in \text{SE}(3)$ 만큼 움직여서 카메라가 $\{C_1\} \rightarrow \{C_2\}$ 으로 이동한 경우를 생각해보자.

\mathbf{X}_1 을 두 번째 카메라 상에 프로젝션한 점 \mathbf{p}_2 는 다음과 같은 공식으로 표현할 수 있다.

$$\mathbf{p}_2 = \pi(\exp(\xi_{21}^\wedge)\mathbf{X}_1) = \pi(\exp(\xi_{21}^\wedge)\pi^{-1}(\mathbf{p}_1))$$

\mathbf{T}_i	카메라의 3차원 포즈 (4x4행렬)
$\xi_{21} \in \mathbb{R}^6$	C1,C2 카메라 간 상대 포즈 변화량 (twist) (6차원벡터)
$\xi_{21}^\wedge \in \text{se}(3)$	hat 연산자가 적용된 twist (lie algebra) (4x4행렬)
$\exp(\xi_{21}^\wedge) \in \text{SE}(3)$	C1,C2 카메라 간 상대 포즈 (transform) (lie group) (4x4행렬)

1.1. Initialization → Error Function Formulation



3차원 상의 한 점 \mathbf{X}_1 과 2차원 이미지 평면 상의 한 점 \mathbf{p}_1 은 다음과 같은 관계를 가진다.

$$\mathbf{p}_1 = \pi(\mathbf{X}_1) = \mathbf{K}\rho_1\mathbf{X}_1$$

where, $\rho_1 = 1/Z_1$ (inverse depth)

$$\mathbf{X}_1 = \pi^{-1}(\mathbf{p}_1) = \mathbf{K}^{-1}/\rho_1\mathbf{p}_1$$

이 때, 카메라가 $\exp(\xi_{21}^\wedge) \in \text{SE}(3)$ 만큼 움직여서 카메라가 $\{C_1\} \rightarrow \{C_2\}$ 으로 이동한 경우를 생각해보자.

\mathbf{X}_1 을 두 번째 카메라 상에 프로젝션한 점 \mathbf{p}_2 는 다음과 같은 공식으로 표현할 수 있다.

$$\mathbf{p}_2 = \pi(\exp(\xi_{21}^\wedge)\mathbf{X}_1) = \pi(\exp(\xi_{21}^\wedge)\pi^{-1}(\mathbf{p}_1))$$

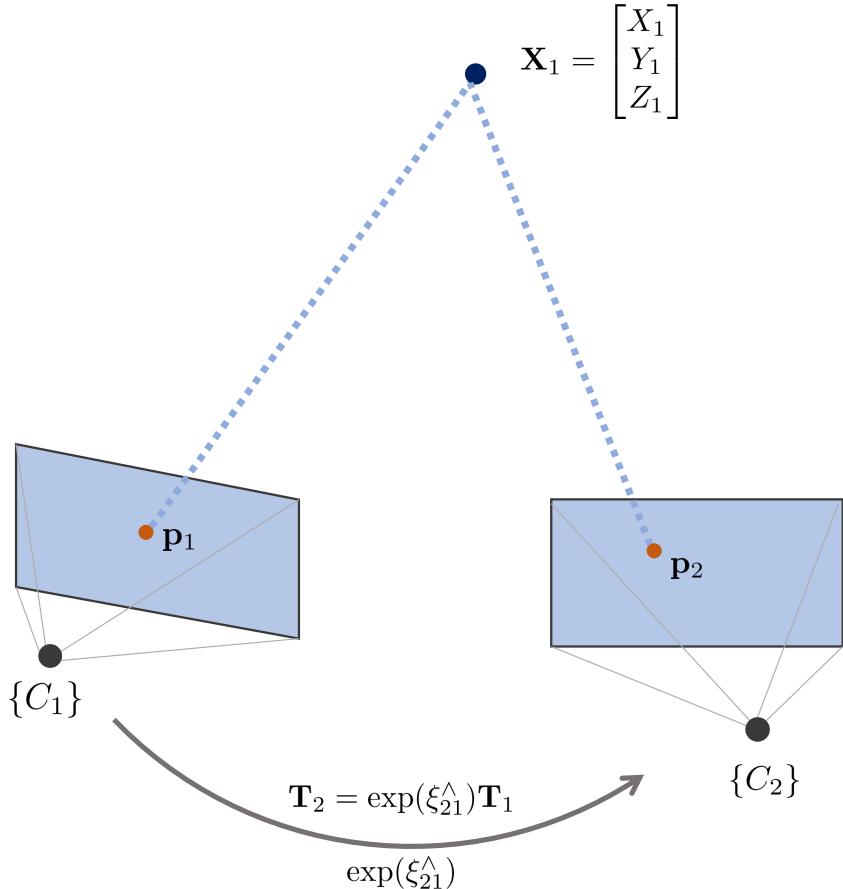
direct method는 photometric error를 에러함수로 사용하므로 픽셀의 밝기(intensity) 값의 차이를 구해야 한다. 이 때 DSO는 카메라의 exposure time을 고려하여 밝기 오차를 조정하는 파라미터를 사용한다. 이를 Photometric Calibration parameter(a, b)라고 한다.

또는 Affine brightness transfer parameters (a, b)

$$\mathbf{I}_2(\mathbf{p}_2) = \exp(a)\mathbf{I}_1(\mathbf{p}_1) + b \quad \text{where, } a = \frac{t_j \exp(a_j)}{t_i \exp(a_i)} \\ b = b_i - b_j$$

$\mathbf{I}_i(\mathbf{p}_j)$	i번째 이미지에서 j번째 점의 밝기 (grayscale)(intensity)(0~255단위)
t_i	i번째 이미지가 촬영된 순간의 exposure time (또는 Shutter time)(ms단위)
a, b	두 이미지 간 exposure time을 고려하여 이미지 밝기를 조절하는 파라미터

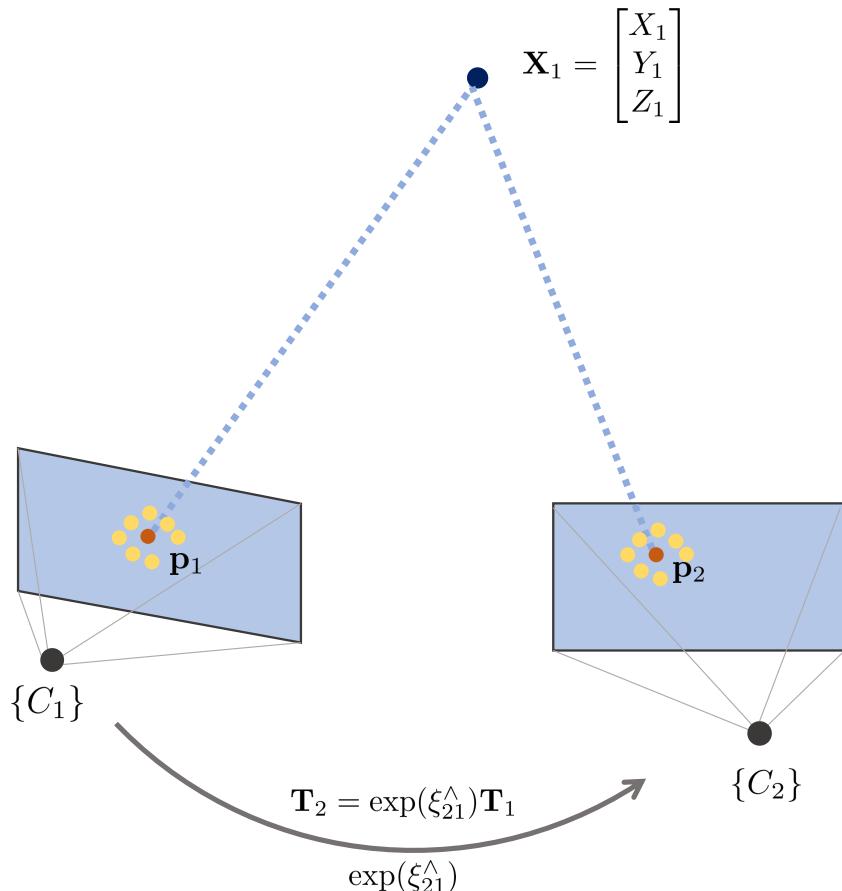
1.1. Initialization → Error Function Formulation



이 때 두 점의 픽셀 밝기의 차이를 **에러(error, residual)**로 설정한다.

$$\mathbf{r}(\mathbf{p}_1) = \mathbf{I}_2(\mathbf{p}_2) - \exp(a)\mathbf{I}_1(\mathbf{p}_1) - b$$

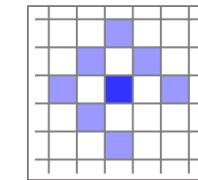
1.1. Initialization → Error Function Formulation



이 때 두 점의 픽셀 밝기의 차이를 **에러(error, residual)**로 설정한다.

$$\mathbf{r}(\mathbf{p}_1) = \mathbf{I}_2(\mathbf{p}_2) - \exp(a)\mathbf{I}_1(\mathbf{p}_1) - b$$

DSO는 한 점에서 밝기 차이만을 고려하지 않고 한 점 주위의 총 8개 점들 patch의 밝기 차이를 고려한다. 아래 이미지와 같이 1개의 점이 생략된 이유는 성능에 크게 영향을 주지 않으면서 4번의 float 연산을 한 번에 계산할 수 있는 SSE2 레지스터를 사용하여 속도를 가속하기 위해서다.



1.1. Initialization → Error Function Formulation

\mathbf{E}_p	두 점 $\mathbf{p}_1, \mathbf{p}_2$ 사이의 밝기 오차 또는 에너지. Photometric Error(Energy)
$\mathcal{N}(\mathbf{p})$	Patch 상의 한 점
w_p	임의의 weighting 함수.
$H(\mathbf{r})$	Huber function
$\ \cdot\ _\gamma$	Huber norm (Huber function과 동일)

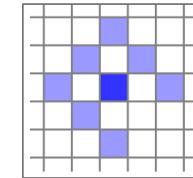
$$\mathbf{X}_1 = \begin{bmatrix} X_1 \\ Y_1 \\ Z_1 \end{bmatrix}$$



이 때 두 점의 픽셀 밝기의 차이를 **에러(error, residual)**로 설정한다.

$$\mathbf{r}(\mathbf{p}_1) = \mathbf{I}_2(\mathbf{p}_2) - \exp(a)\mathbf{I}_1(\mathbf{p}_1) - b$$

DSO는 한 점에서 밝기 차이만을 고려하지 않고 한 점 주위의 총 8개 점들 patch의 밝기 차이를 고려한다. 아래 이미지와 같이 1개의 점이 생략된 이유는 성능에 크게 영향을 주지 않으면서 4번의 float 연산을 한 번에 계산할 수 있는 SSE2 레지스터를 사용하여 속도를 가속하기 위해서다.



patch를 고려하고 weighting 함수 + Huber 함수까지 적용한 에러 함수 \mathbf{E}_p 는 다음과 같다.

$$\mathbf{E}_p = \sum_{\mathbf{p}_1 \in \mathcal{N}(\mathbf{p})} w_p \|\mathbf{I}_2(\mathbf{p}_2) - \exp(a)\mathbf{I}_1(\mathbf{p}_1) - b\|_\gamma = \sum_{\mathbf{p}_1 \in \mathcal{N}(\mathbf{p})} w_p H(\mathbf{r}(\mathbf{p}_1))$$

… Eq. (4) in the paper.

1.1. Initialization → Error Function Formulation

\mathbf{E}_p	두 점 $\mathbf{p}_1, \mathbf{p}_2$ 사이의 밝기 오차 또는 에너지. Photometric Error(Energy)
$\mathcal{N}(\mathbf{p})$	Patch 상의 한 점
w_p	임의의 weighting 함수.
$H(\mathbf{r})$	Huber function
$\ \cdot\ _\gamma$	Huber norm (Huber function과 동일)

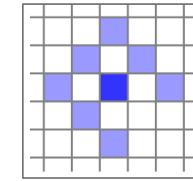
$$\mathbf{X}_1 = \begin{bmatrix} X_1 \\ Y_1 \\ Z_1 \end{bmatrix}$$



이 때 두 점의 픽셀 밝기의 차이를 **에러(error, residual)**로 설정한다.

$$\mathbf{r}(\mathbf{p}_1) = \mathbf{I}_2(\mathbf{p}_2) - \exp(a)\mathbf{I}_1(\mathbf{p}_1) - b$$

DSO는 한 점에서 밝기 차이만을 고려하지 않고 한 점 주위의 총 8개 점들 patch의 밝기 차이를 고려한다. 아래 이미지와 같이 1개의 점이 생략된 이유는 성능에 크게 영향을 주지 않으면서 4번의 float 연산을 한 번에 계산할 수 있는 SSE2 레지스터를 사용하여 속도를 가속하기 위해서다.



patch를 고려하고 weighting 함수 + Huber 함수까지 적용한 에러 함수 \mathbf{E}_p 는 다음과 같다.

$$\mathbf{E}_p = \sum_{\mathbf{p}_1 \in \mathcal{N}(\mathbf{p})} w_p \|\mathbf{I}_2(\mathbf{p}_2) - \exp(a)\mathbf{I}_1(\mathbf{p}_1) - b\|_\gamma = \sum_{\mathbf{p}_1 \in \mathcal{N}(\mathbf{p})} w_p H(\mathbf{r}(\mathbf{p}_1))$$

… Eq. (4) in the paper.

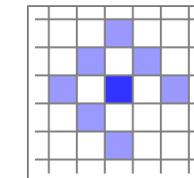
1.1. Initialization → Error Function Formulation

\mathbf{E}_p	두 점 $\mathbf{p}_1, \mathbf{p}_2$ 사이의 밝기 오차 또는 에너지. Photometric Error(Energy)
$\mathcal{N}(\mathbf{p})$	Patch 상의 한 점
w_p	임의의 weighting 함수.
$H(\mathbf{r})$	Huber function
$\ \cdot\ _\gamma$	Huber norm (Huber function과 동일)

이 때 두 점의 픽셀 밝기의 차이를 **에러(error, residual)**로 설정한다.

$$\mathbf{r}(\mathbf{p}_1) = \mathbf{I}_2(\mathbf{p}_2) - \exp(a)\mathbf{I}_1(\mathbf{p}_1) - b$$

DSO는 한 점에서 밝기 차이만을 고려하지 않고 한 점 주위의 총 8개 점들 patch의 밝기 차이를 고려한다. 아래 이미지와 같이 1개의 점이 생략된 이유는 성능에 크게 영향을 주지 않으면서 4번의 float 연산을 한 번에 계산할 수 있는 SSE2 레지스터를 사용하여 속도를 가속하기 위해서다.



patch를 고려하고 weighting 함수 + Huber 함수까지 적용한 에러 함수 \mathbf{E}_p 는 다음과 같다.

$$\mathbf{E}_p = \sum_{\mathbf{p}_1 \in \mathcal{N}(\mathbf{p})} w_p \|\mathbf{I}_2(\mathbf{p}_2) - \exp(a)\mathbf{I}_1(\mathbf{p}_1) - b\|_\gamma = \sum_{\mathbf{p}_1 \in \mathcal{N}(\mathbf{p})} w_p H(\mathbf{r}(\mathbf{p}_1))$$

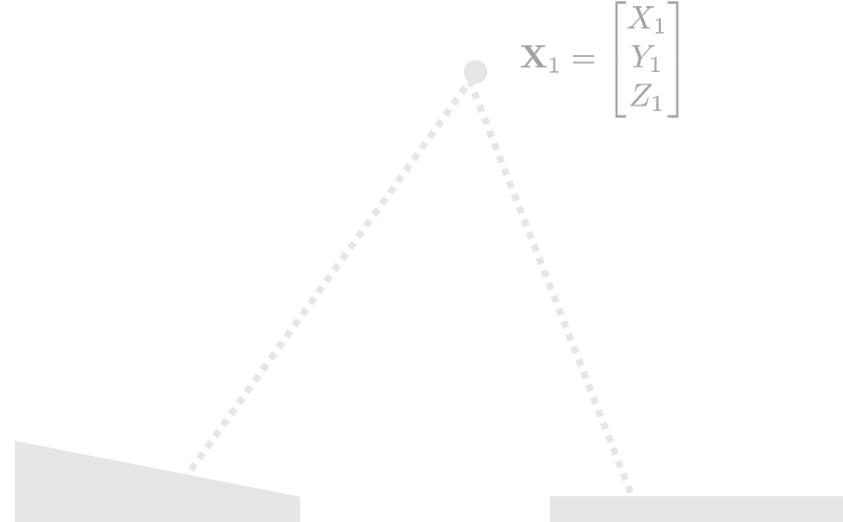
… Eq. (4) in the paper.

임의의 weighting 함수 w_p 는 다음과 같이 정의되고 image gradient가 큰 점의 경우 down-weighting 된다.

$$w_p = \frac{c^2}{c^2 + \|\nabla \mathbf{I}(\mathbf{p})\|_2^2}$$

… Eq. (7) in the paper.

1.1. Initialization → Error Function Formulation



E_p	두 점 $\mathbf{p}_1, \mathbf{p}_2$ 사이의 밝기 오차 또는 에너지. Photometric Error(Energy)
$\mathcal{N}(\mathbf{p})$	Patch 상의 한 점
w_p	임의의 weighting 함수.
$H(\mathbf{r})$	Huber function
$\ \cdot\ _\gamma$	Huber norm (Huber function과 동일)

Huber function $H(\mathbf{r})$ 는 다음과 같이 정의되고 DSO의 경우 $\sigma = 9$ 의 상수값을 가진다.

$$H(\mathbf{r}) = \begin{cases} \mathbf{r}^2/2, & |\mathbf{r}| < \sigma \\ \sigma(|\mathbf{r}| - \sigma/2) & |\mathbf{r}| \geq \sigma \end{cases}$$

1.1. Initialization → Error Function Formulation

Huber function $H(\mathbf{r})$ 는 다음과 같이 정의되고 DSO의 경우 $\sigma = 9$ 의 상수값을 가진다.

$$\mathbf{X}_1 = \begin{bmatrix} X_1 \\ Y_1 \\ Z_1 \end{bmatrix}$$

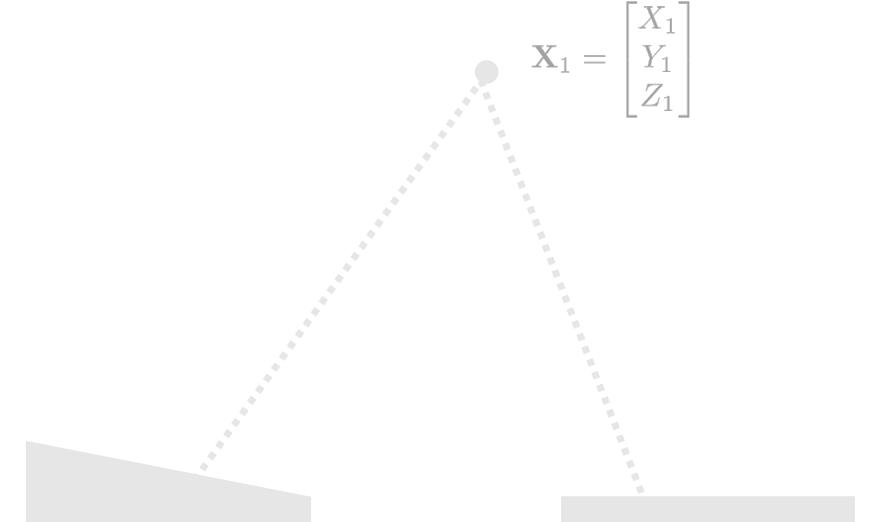
$$H(\mathbf{r}) = \begin{cases} \mathbf{r}^2/2, & |\mathbf{r}| < \sigma \\ \sigma(|\mathbf{r}| - \sigma/2) & |\mathbf{r}| \geq \sigma \end{cases}$$

위 식을 컴팩트하게 표현하기 위해 $w_h = \begin{cases} 1 & |\mathbf{r}| < \sigma \\ \sigma/|\mathbf{r}| & |\mathbf{r}| \geq \sigma \end{cases}$ 를 사용하면 아래와 같이 나타낼 수 있다.

$$H(\mathbf{r}) = w_h \mathbf{r}^2 (1 - w_h/2)$$

E_p	두 점 $\mathbf{p}_1, \mathbf{p}_2$ 사이의 밝기 오차 또는 에너지. Photometric Error(Energy)
$\mathcal{N}(\mathbf{p})$	Patch 상의 한 점
w_p	임의의 weighting 함수.
$H(\mathbf{r})$	Huber function
$\ \cdot\ _\gamma$	Huber norm (Huber function과 동일)

1.1. Initialization → Error Function Formulation



\mathbf{E}_p	두 점 $\mathbf{p}_1, \mathbf{p}_2$ 사이의 밝기 오차 또는 에너지. Photometric Error(Energy)
$\mathcal{N}(\mathbf{p})$	Patch 상의 한 점
w_p	임의의 weighting 함수.
$H(\mathbf{r})$	Huber function
$\ \cdot\ _\gamma$	Huber norm (Huber function과 동일)

Huber function $H(\mathbf{r})$ 는 다음과 같이 정의되고 DSO의 경우 $\sigma = 9$ 의 상수값을 가진다.

$$H(\mathbf{r}) = \begin{cases} \mathbf{r}^2/2, & |\mathbf{r}| < \sigma \\ \sigma(|\mathbf{r}| - \sigma/2) & |\mathbf{r}| \geq \sigma \end{cases}$$

위 식을 컴팩트하게 표현하기 위해 $w_h = \begin{cases} 1 & |\mathbf{r}| < \sigma \\ \sigma/|\mathbf{r}| & |\mathbf{r}| \geq \sigma \end{cases}$ 를 사용하면 아래와 같이 나타낼 수 있다.

$$H(\mathbf{r}) = w_h \mathbf{r}^2 (1 - w_h/2)$$

이를 두 카메라 이미지 상의 모든 점에 대해 에러를 계산하면 아래와 같다.

$$\mathbf{E} = \sum \mathbf{E}_p$$

1.2. Initialization → Gauss-Newton Optimization

Huber function까지 적용한 에러 함수 (또는 residual) $H(\mathbf{r})$ 은 다음과 같다.

$$H(\mathbf{r}) = w_h \mathbf{r}^2 (1 - w_h/2)$$

1.2. Initialization → Gauss-Newton Optimization

Huber function까지 적용한 에러 함수 (또는 residual) $H(\mathbf{r})$ 은 다음과 같다.

$$H(\mathbf{r}) = w_h \mathbf{r}^2 (1 - \frac{w_h}{2})$$

위 함수를 Least Square method로 최적화하기 위해서 $H(\mathbf{r})$ 가 Squared form이라고 가정하면

$f(\mathbf{x})^T f(\mathbf{x}) = H(\mathbf{r})$ 을 만족하는 $f(\mathbf{x})$ 를 생각할 수 있고 이 때 $f(\mathbf{x})$ 가 최소이면 $f(\mathbf{x})$ 의 제곱인 $H(\mathbf{r})$ 역시 최소가 된다.

DSO에서 $H(\mathbf{r})$ 의 $-w_h/2$ 부분은 크기가 상대적으로 작으므로 생략했다.

$$f(\mathbf{x})^T f(\mathbf{x}) = H(\mathbf{r})$$

$$f(\mathbf{x}) = \sqrt{w_h} \mathbf{r} = \sqrt{w_h} (\mathbf{I}_2(\mathbf{p}_2) - \exp(a) \mathbf{I}_1(\mathbf{p}_1) - b)$$

1.2. Initialization → Gauss-Newton Optimization

Huber function까지 적용한 에러 함수 (또는 residual) $H(\mathbf{r})$ 은 다음과 같다.

$$H(\mathbf{r}) = w_h \mathbf{r}^2 (1 - w_h/2) = w_h \mathbf{r}^2$$

위 함수를 Least Square method로 최적화하기 위해서 $H(\mathbf{r})$ 가 Squared form이라고 가정하면

$f(\mathbf{x})^T f(\mathbf{x}) = H(\mathbf{r})$ 을 만족하는 $f(\mathbf{x})$ 를 생각할 수 있고 이 때 $f(\mathbf{x})$ 가 최소이면 $f(\mathbf{x})$ 의 제곱인 $H(\mathbf{r})$ 역시 최소가 된다.

DSO에서 $H(\mathbf{r})$ 의 $-w_h/2$ 부분은 크기가 상대적으로 작으므로 생략했다.

$$f(\mathbf{x})^T f(\mathbf{x}) = H(\mathbf{r})$$

$$f(\mathbf{x}) = \sqrt{w_h} \mathbf{r} = \sqrt{w_h} (\mathbf{I}_2(\mathbf{p}_2) - \exp(a) \mathbf{I}_1(\mathbf{p}_1) - b)$$

따라서 해당 문제를 $f(\mathbf{x})$ 최소화 문제로 변경할 수 있고 이 때 state variable \mathbf{X} 는 다음과 같다.

$$\mathbf{x} = [\rho_1^{(1)}, \dots, \rho_1^{(N)}, \delta\xi^T, a, b]_{(N+8) \times 1}^T$$

$\rho_1^{(i)}$	1번 이미지에서 i번째 inverse depth 값 (N개)
$\delta\xi$	두 카메라 간 상대 포즈 변화량 (twist) (6차원 벡터) (6개)
a, b	두 이미지 간 exposure time을 고려하여 이미지 밝기를 조절하는 파라미터 (2개)

1.2. Initialization → Gauss-Newton Optimization

Huber function까지 적용한 에러 함수 (또는 residual) $H(\mathbf{r})$ 은 다음과 같다.

$$H(\mathbf{r}) = w_h \mathbf{r}^2 (1 - w_h/2) = w_h \mathbf{r}^2$$

위 함수를 Least Square method로 최적화하기 위해서 $H(\mathbf{r})$ 가 Squared form이라고 가정하면

$f(\mathbf{x})^T f(\mathbf{x}) = H(\mathbf{r})$ 을 만족하는 $f(\mathbf{x})$ 를 생각할 수 있고 이 때 $f(\mathbf{x})$ 가 최소이면 $f(\mathbf{x})$ 의 제곱인 $H(\mathbf{r})$ 역시 최소가 된다.

DSO에서 $H(\mathbf{r})$ 의 $-w_h/2$ 부분은 크기가 상대적으로 작으므로 생략했다.

$$f(\mathbf{x})^T f(\mathbf{x}) = H(\mathbf{r})$$

$$f(\mathbf{x}) = \sqrt{w_h} \mathbf{r} = \sqrt{w_h} (\mathbf{I}_2(\mathbf{p}_2) - \exp(a) \mathbf{I}_1(\mathbf{p}_1) - b)$$

따라서 해당 문제를 $f(\mathbf{x})$ 최소화 문제로 변경할 수 있고 이 때 state variable \mathbf{X} 는 다음과 같다.

$$\mathbf{x} = [\rho_1^{(1)}, \dots, \rho_1^{(N)}, \delta\xi^T, a, b]_{(N+8) \times 1}^T$$

$\rho_1^{(i)}$	1번 이미지에서 i번째 inverse depth 값 (N개)
$\delta\xi$	두 카메라 간 상대 포즈 변화량 (twist)(6차원 벡터) (6개)
a, b	두 이미지 간 exposure time을 고려하여 이미지 밝기를 조절하는 파라미터 (2개)

1.2. Initialization → Gauss-Newton Optimization

$f(\mathbf{x})$ 는 비선형 함수이므로 non linear least square method를 사용하여 최소화해야 한다. 이를 위해 **Gauss-Newton method**가 사용된다.

Gauss-Newton method는 반복적(Iterative)으로 $f(\mathbf{x} + \Delta\mathbf{x})$ 값이 감소하는 $\Delta\mathbf{x}$ 를 구하여 비선형 함수를 최적화하는 방법 중 하나이다.

1.2. Initialization → Gauss-Newton Optimization

$f(\mathbf{x})$ 는 비선형 함수이므로 non linear least square method를 사용하여 최소화해야 한다. 이를 위해 **Gauss-Newton method**가 사용된다.

Gauss-Newton method는 반복적(Iterative)으로 $f(\mathbf{x} + \Delta\mathbf{x})$ 값이 감소하는 $\Delta\mathbf{x}$ 를 구하여 비선형 함수를 최적화하는 방법 중 하나이다.

자세한 과정은 다음과 같다.

1. $f(\mathbf{x} + \Delta\mathbf{x})$ 를 1차 테일러 전개하면 다음과 같이 근사할 수 있다.

$$f(\mathbf{x} + \Delta\mathbf{x}) \simeq f(\mathbf{x}) + \mathbf{J}\Delta\mathbf{x}$$

1.2. Initialization → Gauss-Newton Optimization

$f(\mathbf{x})$ 는 비선형 함수이므로 non linear least square method를 사용하여 최소화해야 한다. 이를 위해 **Gauss-Newton method**가 사용된다.

Gauss-Newton method는 반복적(Iterative)으로 $f(\mathbf{x} + \Delta\mathbf{x})$ 값이 감소하는 $\Delta\mathbf{x}$ 를 구하여 비선형 함수를 최적화하는 방법 중 하나이다.

자세한 과정은 다음과 같다.

1. $f(\mathbf{x} + \Delta\mathbf{x})$ 를 1차 테일러 전개하면 다음과 같이 근사할 수 있다.

$$f(\mathbf{x} + \Delta\mathbf{x}) \simeq f(\mathbf{x}) + \mathbf{J}\Delta\mathbf{x}$$

2. $\frac{1}{2}f^2(\mathbf{x} + \Delta\mathbf{x})$ 의 최소값을 구한다고 가정하면 해당 함수의 1차 미분이 0이 되는 $\Delta\mathbf{x}$ 를 구하면 그 점이 최소가 된다.

$$\begin{aligned} f(\mathbf{x} + \Delta\mathbf{x})^T f(\mathbf{x} + \Delta\mathbf{x}) &\simeq (f(\mathbf{x}) + \mathbf{J}\Delta\mathbf{x})^T (f(\mathbf{x}) + \mathbf{J}\Delta\mathbf{x}) \\ &= \underbrace{f(\mathbf{x})^T f(\mathbf{x})}_{\mathbf{c}} + 2\underbrace{f(\mathbf{x})^T \mathbf{J}\Delta\mathbf{x}}_{\mathbf{b}} + \underbrace{\Delta\mathbf{x}^T \mathbf{J}^T \mathbf{J}\Delta\mathbf{x}}_{\mathbf{H}} \\ &= \mathbf{c} + 2\mathbf{b}\Delta\mathbf{x} + \Delta\mathbf{x}^T \mathbf{H}\Delta\mathbf{x} \end{aligned}$$

1.2. Initialization → Gauss-Newton Optimization

$f(\mathbf{x})$ 는 비선형 함수이므로 non linear least square method를 사용하여 최소화해야 한다. 이를 위해 **Gauss-Newton method**가 사용된다.

Gauss-Newton method는 반복적(Iterative)으로 $f(\mathbf{x} + \Delta\mathbf{x})$ 값이 감소하는 $\Delta\mathbf{x}$ 를 구하여 비선형 함수를 최적화하는 방법 중 하나이다.

자세한 과정은 다음과 같다.

1. $f(\mathbf{x} + \Delta\mathbf{x})$ 를 1차 테일러 전개하면 다음과 같이 근사할 수 있다.

$$f(\mathbf{x} + \Delta\mathbf{x}) \simeq f(\mathbf{x}) + \mathbf{J}\Delta\mathbf{x}$$

2. $\frac{1}{2}f^2(\mathbf{x} + \Delta\mathbf{x})$ 의 최소값을 구한다고 가정하면 해당 함수의 1차 미분이 0이 되는 $\Delta\mathbf{x}$ 를 구하면 그 점이 최소가 된다.

$$\begin{aligned} f(\mathbf{x} + \Delta\mathbf{x})^T f(\mathbf{x} + \Delta\mathbf{x}) &\simeq (f(\mathbf{x}) + \mathbf{J}\Delta\mathbf{x})^T (f(\mathbf{x}) + \mathbf{J}\Delta\mathbf{x}) \\ &= \underbrace{f(\mathbf{x})^T f(\mathbf{x})}_{\mathbf{c}} + 2\underbrace{f(\mathbf{x})^T \mathbf{J}\Delta\mathbf{x}}_{\mathbf{b}} + \Delta\mathbf{x}^T \underbrace{\mathbf{J}^T \mathbf{J}}_{\mathbf{H}} \Delta\mathbf{x} \\ &= \mathbf{c} + 2\mathbf{b}\Delta\mathbf{x} + \Delta\mathbf{x}^T \mathbf{H}\Delta\mathbf{x} \end{aligned}$$

1.2. Initialization → Gauss-Newton Optimization

3. $\frac{1}{2}f^2(\mathbf{x} + \Delta\mathbf{x})$ 의 최소값은 다음과 같이 구할 수 있다.

$$\frac{\partial \frac{1}{2}f^2(\mathbf{x} + \Delta\mathbf{x})}{\partial \Delta\mathbf{x}} = f(\mathbf{x} + \Delta\mathbf{x}) \frac{\partial f(\mathbf{x} + \Delta\mathbf{x})}{\partial \Delta\mathbf{x}} \simeq (f(\mathbf{x}) + \mathbf{J}\Delta\mathbf{x})\mathbf{J} = 0$$

1.2. Initialization → Gauss-Newton Optimization

3. $\frac{1}{2}f^2(\mathbf{x} + \Delta\mathbf{x})$ 의 최소값은 다음과 같이 구할 수 있다.

$$\frac{\partial \frac{1}{2}f^2(\mathbf{x} + \Delta\mathbf{x})}{\partial \Delta\mathbf{x}} = f(\mathbf{x} + \Delta\mathbf{x}) \frac{\partial f(\mathbf{x} + \Delta\mathbf{x})}{\partial \Delta\mathbf{x}} \simeq (f(\mathbf{x}) + \mathbf{J}\Delta\mathbf{x})\mathbf{J} = 0$$

1.2. Initialization → Gauss-Newton Optimization

3. $\frac{1}{2}f^2(\mathbf{x} + \Delta\mathbf{x})$ 의 최소값은 다음과 같이 구할 수 있다.

$$\frac{\partial \frac{1}{2}f^2(\mathbf{x} + \Delta\mathbf{x})}{\partial \Delta\mathbf{x}} = f(\mathbf{x} + \Delta\mathbf{x}) \frac{\partial f(\mathbf{x} + \Delta\mathbf{x})}{\partial \Delta\mathbf{x}} \simeq (f(\mathbf{x}) + \mathbf{J}\Delta\mathbf{x})\mathbf{J} = 0$$

4. 위 식을 정리하면 다음과 같은 식이 나오고

$$\mathbf{J}^T \mathbf{J} \Delta\mathbf{x} = -\mathbf{J}^T f(\mathbf{x})$$

1.2. Initialization → Gauss-Newton Optimization

3. $\frac{1}{2}f^2(\mathbf{x} + \Delta\mathbf{x})$ 의 최소값은 다음과 같이 구할 수 있다.

$$\frac{\partial \frac{1}{2}f^2(\mathbf{x} + \Delta\mathbf{x})}{\partial \Delta\mathbf{x}} = f(\mathbf{x} + \Delta\mathbf{x}) \frac{\partial f(\mathbf{x} + \Delta\mathbf{x})}{\partial \Delta\mathbf{x}} \simeq (f(\mathbf{x}) + \mathbf{J}\Delta\mathbf{x})\mathbf{J} = 0$$

4. 위 식을 정리하면 다음과 같은 식이 나오고

$$\mathbf{J}^T \mathbf{J} \Delta\mathbf{x} = -\mathbf{J}^T f(\mathbf{x})$$

1.2. Initialization → Gauss-Newton Optimization

3. $\frac{1}{2}f^2(\mathbf{x} + \Delta\mathbf{x})$ 의 최소값은 다음과 같이 구할 수 있다.

$$\frac{\partial \frac{1}{2}f^2(\mathbf{x} + \Delta\mathbf{x})}{\partial \Delta\mathbf{x}} = f(\mathbf{x} + \Delta\mathbf{x}) \frac{\partial f(\mathbf{x} + \Delta\mathbf{x})}{\partial \Delta\mathbf{x}} \simeq (f(\mathbf{x}) + \mathbf{J}\Delta\mathbf{x})\mathbf{J} = 0$$

4. 위 식을 정리하면 다음과 같은 식이 나오고

$$\mathbf{J}^T \mathbf{J} \Delta\mathbf{x} = -\mathbf{J}^T f(\mathbf{x})$$

5. 이를 컴팩트하게 표현하기 위해 치환하면 다음과 같다.

$$\mathbf{H}\Delta\mathbf{x} = -\mathbf{b}$$

where, $\mathbf{H} = \sum \mathbf{J}^T \mathbf{J}$
 $\mathbf{b} = \sum \mathbf{J}^T f(\mathbf{x})$

1.2. Initialization → Gauss-Newton Optimization

3. $\frac{1}{2}f^2(\mathbf{x} + \Delta\mathbf{x})$ 의 최소값은 다음과 같이 구할 수 있다.

$$\frac{\partial \frac{1}{2}f^2(\mathbf{x} + \Delta\mathbf{x})}{\partial \Delta\mathbf{x}} = f(\mathbf{x} + \Delta\mathbf{x}) \frac{\partial f(\mathbf{x} + \Delta\mathbf{x})}{\partial \Delta\mathbf{x}} \simeq (f(\mathbf{x}) + \mathbf{J}\Delta\mathbf{x})\mathbf{J} = 0$$

4. 위 식을 정리하면 다음과 같은 식이 나오고

$$\mathbf{J}^T \mathbf{J} \Delta\mathbf{x} = -\mathbf{J}^T f(\mathbf{x})$$

5. 이를 컴팩트하게 표현하기 위해 치환하면 다음과 같다.

$$\mathbf{H}\Delta\mathbf{x} = -\mathbf{b}$$
 where, $\mathbf{H} = \sum \mathbf{J}^T \mathbf{J}$
$$\mathbf{b} = \sum \mathbf{J}^T f(\mathbf{x})$$

1.2. Initialization → Gauss-Newton Optimization

3. $\frac{1}{2}f^2(\mathbf{x} + \Delta\mathbf{x})$ 의 최소값은 다음과 같이 구할 수 있다.

$$\frac{\partial \frac{1}{2}f^2(\mathbf{x} + \Delta\mathbf{x})}{\partial \Delta\mathbf{x}} = f(\mathbf{x} + \Delta\mathbf{x}) \frac{\partial f(\mathbf{x} + \Delta\mathbf{x})}{\partial \Delta\mathbf{x}} \simeq (f(\mathbf{x}) + \mathbf{J}\Delta\mathbf{x})\mathbf{J} = 0$$

4. 위 식을 정리하면 다음과 같은 식이 나오고

$$\mathbf{J}^T \mathbf{J} \Delta\mathbf{x} = -\mathbf{J}^T f(\mathbf{x})$$

5. 이를 컴팩트하게 표현하기 위해 치환하면 다음과 같다.

$$\mathbf{H}\Delta\mathbf{x} = -\mathbf{b}$$
 where, $\mathbf{H} = \sum \mathbf{J}^T \mathbf{J}$
$$\mathbf{b} = \sum \mathbf{J}^T f(\mathbf{x})$$

6. 위 식을 계산하면 최적의 $\Delta\mathbf{x}$ 를 계산할 수 있고 이를 기존의 state variable에 업데이트 해준다.

$$\Delta\mathbf{x}^* = -\mathbf{H}^{-1}\mathbf{b}$$

$$\therefore \mathbf{x} \leftarrow \mathbf{x} + \Delta\mathbf{x}^*$$

1.2. Initialization → Gauss-Newton Optimization

3. $\frac{1}{2}f^2(\mathbf{x} + \Delta\mathbf{x})$ 의 최소값은 다음과 같이 구할 수 있다.

$$\frac{\partial \frac{1}{2}f^2(\mathbf{x} + \Delta\mathbf{x})}{\partial \Delta\mathbf{x}} = f(\mathbf{x} + \Delta\mathbf{x}) \frac{\partial f(\mathbf{x} + \Delta\mathbf{x})}{\partial \Delta\mathbf{x}} \simeq (f(\mathbf{x}) + \mathbf{J}\Delta\mathbf{x})\mathbf{J} = 0$$

4. 위 식을 정리하면 다음과 같은 식이 나오고

$$\mathbf{J}^T \mathbf{J} \Delta\mathbf{x} = -\mathbf{J}^T f(\mathbf{x})$$

5. 이를 컴팩트하게 표현하기 위해 치환하면 다음과 같다.

$$\mathbf{H}\Delta\mathbf{x} = -\mathbf{b}$$
 where, $\mathbf{H} = \sum \mathbf{J}^T \mathbf{J}$
$$\mathbf{b} = \sum \mathbf{J}^T f(\mathbf{x})$$

6. 위 식을 계산하면 최적의 $\Delta\mathbf{x}$ 를 계산할 수 있고 이를 기존의 state variable에 업데이트 해준다.

$$\Delta\mathbf{x}^* = -\mathbf{H}^{-1}\mathbf{b}$$

$$\therefore \mathbf{x} \leftarrow \mathbf{x} + \Delta\mathbf{x}^*$$

1.3. Initialization → Jacobian Derivation

앞서 언급한 Gauss Newton method를 사용하여 최적화를 하기 위해서는 Jacobian \mathbf{J} 계산이 선행되어야 한다.

$$\boxed{\therefore f(\mathbf{x} + \Delta\mathbf{x}) \simeq f(\mathbf{x}) + \mathbf{J}\Delta\mathbf{x}}$$

1.3. Initialization → Jacobian Derivation

앞서 언급한 Gauss Newton method를 사용하여 최적화를 하기 위해서는 Jacobian \mathbf{J} 계산이 선행되어야 한다.

$$\therefore f(\mathbf{x} + \Delta\mathbf{x}) \simeq f(\mathbf{x}) + \mathbf{J}\Delta\mathbf{x}$$

$f(\mathbf{x})$ 의 state variable \mathbf{x} 를 다음과 같이 정의했으므로 총 $N+8$ 개의 변수에 대한 Jacobian을 계산해야 한다. (inverse depth(N), relative pose(6), photometric parameters(2))

$$\mathbf{x} = [\rho_1^{(1)}, \dots, \rho_1^{(N)}, \delta\xi^T, a, b]_{(N+8) \times 1}^T$$

$$\rho_1^{(i)}$$

1번 이미지에서 i번째 inverse depth 값 (**N 개**)

두 카메라 간 상대 포즈 변화량 (twist)(6차원 벡터) (**6개**)

$$a, b$$

두 이미지 간 exposure time을 고려하여 이미지 밝기를 조절하는 파라미터 (**2개**)

1.3. Initialization → Jacobian Derivation

앞서 언급한 Gauss Newton method를 사용하여 최적화를 하기 위해서는 Jacobian \mathbf{J} 계산이 선행되어야 한다.

$$\therefore f(\mathbf{x} + \Delta\mathbf{x}) \simeq f(\mathbf{x}) + \mathbf{J}\Delta\mathbf{x}$$

$f(\mathbf{x})$ 의 state variable \mathbf{x} 를 다음과 같이 정의했으므로 총 $N+8$ 개의 변수에 대한 Jacobian을 계산해야 한다. (inverse depth(N), relative pose(6), photometric parameters(2))

$$\mathbf{x} = [\rho_1^{(1)}, \dots, \rho_1^{(N)}, \delta\xi^T, a, b]_{(N+8) \times 1}^T$$

$$\rho_1^{(i)}$$

1번 이미지에서 i번째 inverse depth 값 (N 개)

두 카메라 간 상대 포즈 변화량 (twist)(6차원 벡터) (6개)

$$a, b$$

두 이미지 간 exposure time을 고려하여 이미지 밝기를 조절하는 파라미터 (2개)

derivation of photometric parameters

photometric parameter a, b 에 대한 Jacobian은 다음과 같이 계산할 수 있다.

photometric(2)

$$\frac{\partial f(\mathbf{x})}{\partial a} = -\sqrt{w_h} \exp(a) \mathbf{I}_1(\mathbf{p}_1)$$

$$\frac{\partial f(\mathbf{x})}{\partial b} = -\sqrt{w_h}$$

where, $f(\mathbf{x}) = \sqrt{w_h} \mathbf{r} = \sqrt{w_h} (\mathbf{I}_2(\mathbf{p}_2) - \exp(a) \mathbf{I}_1(\mathbf{p}_1) - b)$

1.3. Initialization → Jacobian Derivation

derivation of relative pose

에러 함수 $f(\mathbf{x})$ 가 카메라 포즈 변화량 $\delta\xi$ 에 따라 어떻게 변하는지 계산하기 위해 $\frac{\partial f(\mathbf{x})}{\partial \delta\xi}$ 를 구해야 한다.

1.3. Initialization → Jacobian Derivation

derivation of relative pose

에러 함수 $f(\mathbf{x})$ 가 카메라 포즈 변화량 $\delta\xi$ 에 따라 어떻게 변하는지 계산하기 위해 $\frac{\partial f(\mathbf{x})}{\partial \delta\xi}$ 를 구해야 한다.

다음과 같이 Jacobian 연쇄법칙을 사용하여 $\frac{\partial f(\mathbf{x})}{\partial \delta\xi}$ 를 유도할 수 있다. 이를 위해 개별 Jacobian들을 모두 구해야 한다.

$$\frac{\partial f(\mathbf{x})}{\partial \delta\xi} = \frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2} \frac{\partial \mathbf{p}_2}{\partial \delta\xi} = \frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2} \frac{\partial \mathbf{p}_2}{\partial \mathbf{X}_2} \frac{\partial \mathbf{X}_2}{\partial \delta\xi}$$

1.3. Initialization → Jacobian Derivation

derivation of relative pose

에러 함수 $f(\mathbf{x})$ 가 카메라 포즈 변화량 $\delta\xi$ 에 따라 어떻게 변하는지 계산하기 위해 $\frac{\partial f(\mathbf{x})}{\partial \delta\xi}$ 를 구해야 한다.

다음과 같이 Jacobian 연쇄법칙을 사용하여 $\frac{\partial f(\mathbf{x})}{\partial \delta\xi}$ 를 유도할 수 있다. 이를 위해 개별 Jacobian들을 모두 구해야 한다.

$$\frac{\partial f(\mathbf{x})}{\partial \delta\xi} = \frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2} \frac{\partial \mathbf{p}_2}{\partial \delta\xi} = \frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2} \frac{\partial \mathbf{p}_2}{\partial \mathbf{X}_2} \frac{\partial \mathbf{X}_2}{\partial \delta\xi}$$

1.3. Initialization → Jacobian Derivation

derivation of relative pose

에러 함수 $f(\mathbf{x})$ 가 카메라 포즈 변화량 $\delta\xi$ 에 따라 어떻게 변하는지 계산하기 위해 $\frac{\partial f(\mathbf{x})}{\partial \delta\xi}$ 를 구해야 한다.

다음과 같이 Jacobian 연쇄법칙을 사용하여 $\frac{\partial f(\mathbf{x})}{\partial \delta\xi}$ 를 유도할 수 있다. 이를 위해 개별 Jacobian들을 모두 구해야 한다.

$$\frac{\partial f(\mathbf{x})}{\partial \delta\xi} = \frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2} \frac{\partial \mathbf{p}_2}{\partial \delta\xi} = \frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2} \frac{\partial \mathbf{p}_2}{\partial \mathbf{X}_2} \frac{\partial \mathbf{X}_2}{\partial \delta\xi}$$

$\nabla \mathbf{I}_x, \nabla \mathbf{I}_y$ 를 \mathbf{p}_2 에서 각각 x, y 방향의 image gradient라고 하면 $\frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2}$ 는 다음과 같이 구할 수 있다.

$$\frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2} = \sqrt{w_h} \frac{\partial \mathbf{I}_2(\mathbf{p}_2)}{\partial \mathbf{p}_2} = \sqrt{w_h} [\nabla \mathbf{I}_x \quad \nabla \mathbf{I}_y]$$

where, $f(\mathbf{x}) = \sqrt{w_h} \mathbf{r} = \sqrt{w_h} (\mathbf{I}_2(\mathbf{p}_2) - \exp(a) \mathbf{I}_1(\mathbf{p}_1) - b)$

1.3. Initialization → Jacobian Derivation

$$\frac{\partial f(\mathbf{x})}{\partial \delta \xi} = \frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2} \frac{\partial \mathbf{p}_2}{\partial \delta \xi} = \frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2} \frac{\partial \mathbf{p}_2}{\partial \mathbf{X}_2} \frac{\partial \mathbf{X}_2}{\partial \delta \xi}$$

derivation of relative pose

다음으로 $\frac{\partial \mathbf{p}_2}{\partial \delta \xi} = \frac{\partial \mathbf{p}_2}{\partial \mathbf{X}_2} \frac{\partial \mathbf{X}_2}{\partial \delta \xi}$ 를 구해보면, 우선 $\frac{\partial \mathbf{p}_2}{\partial \mathbf{X}_2}$ 는 2차원 이미지 평면 상의 점 \mathbf{p}_2 와 3차원 공간 상의 점 \mathbf{X}_2 사이의 관계를 의미하므로

1.3. Initialization → Jacobian Derivation

$$\frac{\partial f(\mathbf{x})}{\partial \delta \xi} = \frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2} \frac{\partial \mathbf{p}_2}{\partial \delta \xi} = \frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2} \frac{\partial \mathbf{p}_2}{\partial \mathbf{X}_2} \frac{\partial \mathbf{X}_2}{\partial \delta \xi}$$

derivation of relative pose

다음으로 $\frac{\partial \mathbf{p}_2}{\partial \delta \xi} = \frac{\partial \mathbf{p}_2}{\partial \mathbf{X}_2} \frac{\partial \mathbf{X}_2}{\partial \delta \xi}$ 를 구해보면, 우선 $\frac{\partial \mathbf{p}_2}{\partial \mathbf{X}_2}$ 는 2차원 이미지 평면 상의 점 \mathbf{p}_2 와 3차원 공간 상의 점 \mathbf{X}_2 사이의 관계를 의미하므로

두 점 사이에는 다음과 같은 공식이 성립하고

$$\begin{bmatrix} \mathbf{p}_2 \\ u_2 \\ v_2 \\ 1 \end{bmatrix} = \rho_2 \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{X}_2 \\ X_2 \\ Y_2 \\ Z_2 \end{bmatrix} \quad \text{where, } \rho_2 = \frac{1}{Z_2}, \mathbf{X}_2 = [X_2 \ Y_2 \ Z_2]^T$$

1.3. Initialization → Jacobian Derivation

$$\frac{\partial f(\mathbf{x})}{\partial \delta \xi} = \frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2} \frac{\partial \mathbf{p}_2}{\partial \delta \xi} = \frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2} \frac{\partial \mathbf{p}_2}{\partial \mathbf{X}_2} \frac{\partial \mathbf{X}_2}{\partial \delta \xi}$$

derivation of relative pose

다음으로 $\frac{\partial \mathbf{p}_2}{\partial \delta \xi} = \frac{\partial \mathbf{p}_2}{\partial \mathbf{X}_2} \frac{\partial \mathbf{X}_2}{\partial \delta \xi}$ 를 구해보면, 우선 $\frac{\partial \mathbf{p}_2}{\partial \mathbf{X}_2}$ 는 2차원 이미지 평면 상의 점 \mathbf{p}_2 와 3차원 공간 상의 점 \mathbf{X}_2 사이의 관계를 의미하므로

두 점 사이에는 다음과 같은 공식이 성립하고

$$\begin{bmatrix} \mathbf{p}_2 \\ u_2 \\ v_2 \\ 1 \end{bmatrix} = \rho_2 \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{X}_2 \\ X_2 \\ Y_2 \\ Z_2 \end{bmatrix} \quad \text{where, } \rho_2 = \frac{1}{Z_2}, \mathbf{X}_2 = [X_2 \ Y_2 \ Z_2]^T$$

위 식을 통해 u_2, v_2, X_2, Y_2, Z_2 관계식을 도출할 수 있고 이를 토대로 $\frac{\partial \mathbf{p}_2}{\partial \mathbf{X}_2}$ 를 구해보면 아래와 같다.

$$\frac{\partial \mathbf{p}_2}{\partial \mathbf{X}_2} = \begin{bmatrix} \frac{\partial u_2}{\partial X_2} & \frac{\partial u_2}{\partial Y_2} & \frac{\partial u_2}{\partial Z_2} \\ \frac{\partial v_2}{\partial X_2} & \frac{\partial v_2}{\partial Y_2} & \frac{\partial v_2}{\partial Z_2} \end{bmatrix} = \begin{bmatrix} \rho_2 f_x & 0 & -\rho_2^2 f_x X_2 \\ 0 & \rho_2 f_y & -\rho_2^2 f_y Y_2 \end{bmatrix}$$

1.3. Initialization → Jacobian Derivation

$$\frac{\partial f(\mathbf{x})}{\partial \delta\xi} = \frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2} \frac{\partial \mathbf{p}_2}{\partial \delta\xi} = \frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2} \frac{\partial \mathbf{p}_2}{\partial \mathbf{X}_2} \frac{\partial \mathbf{X}_2}{\partial \delta\xi}$$

derivation of relative pose

다음으로 $\frac{\partial \mathbf{p}_2}{\partial \delta\xi} = \frac{\partial \mathbf{p}_2}{\partial \mathbf{X}_2} \frac{\partial \mathbf{X}_2}{\partial \delta\xi}$ 에서 $\frac{\partial \mathbf{X}_2}{\partial \delta\xi}$ 를 구하기 전 $\delta\xi$ 에 대해 알아보면

$\delta\xi \in \mathbb{R}^6$	카메라 간 상대 포즈 변화량 (twist)(6차원벡터)
$\delta\xi^\wedge \in \text{se}(3)$	hat 연산자가 적용된 twist (4x4행렬)(lie algebra)
$\Delta \mathbf{T} \in \text{SE}(3)$	카메라 간 상대 포즈 (transform)(4x4행렬)(lie group)

lie theory에 대해 보다 자세한 유도 및 설명은 아래 링크 참조

<https://docs.google.com/document/d/1icPiUyT3nPyjZ1OVMtWp9afUtuJ4gXLJI-ex7A9FpNs/edit?usp=sharing>
<http://ethaneade.com/lie.pdf>

1.3. Initialization → Jacobian Derivation

$$\frac{\partial f(\mathbf{x})}{\partial \delta\xi} = \frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2} \frac{\partial \mathbf{p}_2}{\partial \delta\xi} = \frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2} \frac{\partial \mathbf{p}_2}{\partial \mathbf{X}_2} \frac{\partial \mathbf{X}_2}{\partial \delta\xi}$$

derivation of relative pose

다음으로 $\frac{\partial \mathbf{p}_2}{\partial \delta\xi} = \frac{\partial \mathbf{p}_2}{\partial \mathbf{X}_2} \frac{\partial \mathbf{X}_2}{\partial \delta\xi}$ 에서 $\frac{\partial \mathbf{X}_2}{\partial \delta\xi}$ 를 구하기 전 $\delta\xi$ 에 대해 알아보면

$\delta\xi \in \mathbb{R}^6$ 는 3차원 카메라의 포즈 변화량을 의미하고 twist라고 부른다.

$\delta\xi \in \mathbb{R}^6$	카메라 간 상대 포즈 변화량 (twist)(6차원벡터)
$\delta\xi^\wedge \in \text{se}(3)$	hat 연산자가 적용된 twist (4x4행렬)(lie algebra)
$\Delta \mathbf{T} \in \text{SE}(3)$	카메라 간 상대 포즈 (transform)(4x4행렬)(lie group)

lie theory에 대해 보다 자세한 유도 및 설명은 아래 링크 참조

<https://docs.google.com/document/d/1icPiUyT3nPyjZ1OVMtWp9afUtuJ4gXLJI-ex7A9FpNs/edit?usp=sharing>
<http://ethaneade.com/lie.pdf>

1.3. Initialization → Jacobian Derivation

$$\frac{\partial f(\mathbf{x})}{\partial \delta \xi} = \frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2} \frac{\partial \mathbf{p}_2}{\partial \delta \xi} = \frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2} \frac{\partial \mathbf{p}_2}{\partial \mathbf{X}_2} \frac{\partial \mathbf{X}_2}{\partial \delta \xi}$$

derivation of relative pose

다음으로 $\frac{\partial \mathbf{p}_2}{\partial \delta \xi} = \frac{\partial \mathbf{p}_2}{\partial \mathbf{X}_2} \frac{\partial \mathbf{X}_2}{\partial \delta \xi}$ 에서 $\frac{\partial \mathbf{X}_2}{\partial \delta \xi}$ 를 구하기 전 $\delta \xi$ 에 대해 알아보면

$\delta \xi \in \mathbb{R}^6$ 는 3차원 카메라의 포즈 변화량을 의미하고 twist라고 부른다.

$\delta \xi^\wedge \in \text{se}(3)$ 는 twist에 hat operator를 적용한 값으로써 Lie algebra of SE(3)라고 부른다.

위 값에 exponential mapping을 적용하면 4x4 행렬의 3차원 포즈 $\Delta \mathbf{T} \in \text{SE}(3)$ 로 변환할 수 있다. 이를 Lie group of SE(3)라고 부른다.

자세하게 표현하면 다음과 같다.

$\delta \xi \in \mathbb{R}^6$	카메라 간 상대 포즈 변화량 (twist)(6차원벡터)
$\delta \xi^\wedge \in \text{se}(3)$	hat 연산자가 적용된 twist (4x4행렬)(lie algebra)
$\Delta \mathbf{T} \in \text{SE}(3)$	카메라 간 상대 포즈 (transform)(4x4행렬)(lie group)

$$\delta \xi = \begin{bmatrix} \delta \mathbf{v} \\ \delta \mathbf{w} \end{bmatrix} = \begin{bmatrix} \delta v_x \\ \delta v_y \\ \delta v_z \\ \delta w_x \\ \delta w_y \\ \delta w_z \end{bmatrix} \in \mathbb{R}^6$$

linear velocity

angular velocity

$$\delta \xi^\wedge = \begin{bmatrix} \delta \mathbf{w}^\wedge & \delta \mathbf{v} \\ \mathbf{0}^T & 0 \end{bmatrix} \in \text{se}(3) \in \mathbb{R}^{4 \times 4}$$

$$\Delta \mathbf{T} = \exp(\delta \xi^\wedge) = \begin{bmatrix} \Delta \mathbf{R} & \Delta \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix} \in \text{SE}(3) \in \mathbb{R}^{4 \times 4}$$

lie theory에 대해 보다 자세한 유도 및 설명은 아래 링크 참조

<https://docs.google.com/document/d/1icPiUyT3nPyjZ1OVMtWp9afUtuJ4gXLJI-ex7A9FpNs/edit?usp=sharing>
<http://ethaneade.com/lie.pdf>

1.3. Initialization → Jacobian Derivation

$$\frac{\partial f(\mathbf{x})}{\partial \delta \xi} = \frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2} \frac{\partial \mathbf{p}_2}{\partial \delta \xi} = \frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2} \frac{\partial \mathbf{p}_2}{\partial \mathbf{X}_2} \frac{\partial \mathbf{X}_2}{\partial \delta \xi}$$

derivation of relative pose

$\frac{\partial \mathbf{X}_2}{\partial \delta \xi}$ 를 유도해보면 다음과 같다.

$$\begin{aligned}\frac{\partial \mathbf{X}_2}{\partial \delta \xi} &= \lim_{\delta \xi \rightarrow 0} \frac{\exp(\delta \xi \wedge) \mathbf{X}_2 - \mathbf{X}_2}{\delta \xi} \\ &\simeq \lim_{\delta \xi \rightarrow 0} \frac{(\mathbf{I} + \delta \xi \wedge) \mathbf{X}_2 - \mathbf{X}_2}{\delta \xi} \\ &= \lim_{\delta \xi \rightarrow 0} \frac{\delta \xi \wedge \mathbf{X}_2}{\delta \xi} \\ &= \lim_{\delta \xi \rightarrow 0} \frac{\begin{bmatrix} \delta \mathbf{w}^\wedge & \delta \mathbf{v} \\ \mathbf{0}^T & 0 \end{bmatrix} \begin{bmatrix} \mathbf{X}_2 \\ 1 \end{bmatrix}}{\delta \xi} \\ &= \begin{bmatrix} \mathbf{I} & -\mathbf{X}_2^\wedge \\ \mathbf{0}^T & \mathbf{0}^T \end{bmatrix}\end{aligned}$$

$$\therefore \frac{\partial \mathbf{X}_2}{\partial \delta \xi} = [\mathbf{I} \quad -\mathbf{X}_2^\wedge] = \begin{bmatrix} 1 & 0 & 0 & 0 & Z_2 & Y_2 \\ 0 & 1 & 0 & -Z_2 & 0 & X_2 \\ 0 & 0 & 1 & Y_2 & -X_2 & 0 \end{bmatrix}$$

in non-homogeneous form

lie theory에 대해 보다 자세한 유도 및 설명은 아래 링크 참조

<https://docs.google.com/document/d/1icPiUyT3nPyjZ1OVMtWp9afUtuJ4gXLJI-ex7A9FpNs/edit?usp=sharing>
<http://ethaneade.com/lie.pdf>

1.3. Initialization → Jacobian Derivation

derivation of relative pose

지금까지 유도한 공식들을 결합하여 하나의 Jacobian of relative pose로 표현하면 다음과 같다.

$$\boxed{\frac{\partial f(\mathbf{x})}{\partial \delta \xi}} = \frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2} \frac{\partial \mathbf{p}_2}{\partial \delta \xi} = \frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2} \frac{\partial \mathbf{p}_2}{\partial \mathbf{X}_2} \frac{\partial \mathbf{X}_2}{\partial \delta \xi}$$

1.3. Initialization → Jacobian Derivation

derivation of relative pose

지금까지 유도한 공식들을 결합하여 하나의 Jacobian of relative pose로 표현하면 다음과 같다.

$$\boxed{\frac{\partial f(\mathbf{x})}{\partial \delta \xi}} = \frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2} \frac{\partial \mathbf{p}_2}{\partial \delta \xi} = \frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2} \frac{\partial \mathbf{p}_2}{\partial \mathbf{X}_2} \frac{\partial \mathbf{X}_2}{\partial \delta \xi}$$

$$\boxed{\frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2}} = \sqrt{w_h} \frac{\partial \mathbf{I}_2(\mathbf{p}_2)}{\partial \mathbf{p}_2} = \sqrt{w_h} [\nabla \mathbf{I}_x \quad \nabla \mathbf{I}_y]$$

$$\begin{aligned} \frac{\partial \mathbf{p}_2}{\partial \delta \xi} &= \begin{bmatrix} \rho_2 f_x & 0 & -\rho_2^2 f_x X_2 & -\rho_2^2 f_x X_2 Y_2 & f_x + \rho_2^2 f_x X_2^2 & -\rho_2 f_x Y_2 \\ 0 & \rho_2 f_y & -\rho_2^2 f_y Y_2 & -f_y - \rho_2^2 f_y Y_2^2 & \rho_2^2 f_y X_2 Y_2 & \rho_2 f_y X_2 \end{bmatrix} \\ &= \begin{bmatrix} \rho_2 f_x & 0 & -\rho_2^2 f_x u'_2 & -f_x u'_2 v'_2 & f_x + f_x u'^2 & -f_x v'_2 \\ 0 & \rho_2 f_y & -\rho_2^2 f_y v'_2 & -f_y - f_y v'^2 & f_y u'_2 v'_2 & f_y u'_2 \end{bmatrix} \end{aligned}$$

where, $u' = \frac{X_2}{Z_2}, v' = \frac{Y_2}{Z_2}$ in the normalized coordinate.

1.3. Initialization → Jacobian Derivation

derivation of relative pose

지금까지 유도한 공식들을 결합하여 하나의 Jacobian of relative pose로 표현하면 다음과 같다.

$$\frac{\partial f(\mathbf{x})}{\partial \delta \xi} = \frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2} \frac{\partial \mathbf{p}_2}{\partial \delta \xi} = \frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2} \frac{\partial \mathbf{p}_2}{\partial \mathbf{X}_2} \frac{\partial \mathbf{X}_2}{\partial \delta \xi}$$

$$\frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2} = \sqrt{w_h} \frac{\partial \mathbf{I}_2(\mathbf{p}_2)}{\partial \mathbf{p}_2} = \sqrt{w_h} [\nabla \mathbf{I}_x \quad \nabla \mathbf{I}_y]$$

$$\begin{aligned} \frac{\partial \mathbf{p}_2}{\partial \delta \xi} &= \begin{bmatrix} \rho_2 f_x & 0 & -\rho_2^2 f_x X_2 & -\rho_2^2 f_x X_2 Y_2 & f_x + \rho_2^2 f_x X_2^2 & -\rho_2 f_x Y_2 \\ 0 & \rho_2 f_y & -\rho_2^2 f_y Y_2 & -f_y - \rho_2^2 f_y Y_2^2 & \rho_2^2 f_y X_2 Y_2 & \rho_2 f_y X_2 \end{bmatrix} \\ &= \begin{bmatrix} \rho_2 f_x & 0 & -\rho_2^2 f_x u'_2 & -f_x u'_2 v'_2 & f_x + f_x u'^2 & -f_x v'_2 \\ 0 & \rho_2 f_y & -\rho_2^2 f_y v'_2 & -f_y - f_y v'^2 & f_y u'_2 v'_2 & f_y u'_2 \end{bmatrix} \end{aligned}$$

where, $u' = \frac{X_2}{Z_2}, v' = \frac{Y_2}{Z_2}$ in the normalized coordinate.

pose(6)

$$\therefore \frac{\partial f(\mathbf{x})}{\partial \delta \xi} = \sqrt{w_h} \begin{bmatrix} \nabla \mathbf{I}_x \rho_2 f_x \\ \nabla \mathbf{I}_y \rho_2 f_y \\ -\rho_2 (\nabla \mathbf{I}_x f_x u'_2 + \nabla \mathbf{I}_y f_y v'_2) \\ -\nabla \mathbf{I}_x f_x u'_2 v'_2 - \nabla \mathbf{I}_y f_y (1 + v'^2) \\ \nabla \mathbf{I}_x f_x (1 + u'^2) + \nabla \mathbf{I}_y f_y u'_2 v'_2 \\ -\nabla \mathbf{I}_x f_x v'_2 + \nabla \mathbf{I}_y f_y u'_2 \end{bmatrix}^T$$

1.3. Initialization → Jacobian Derivation

derivation of inverse depth

마지막으로 Jacobian of inverse depth $\frac{\partial f(\mathbf{x})}{\partial \rho_1}$ 를 유도해야 한다.

$\frac{\partial f(\mathbf{x})}{\partial \rho_1}$ 는 다음과 같이 여러 연쇄적인 Jacobian의 곱으로 표현할 수 있다.

$$\frac{\partial f(\mathbf{x})}{\partial \rho_1} = \frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2} \frac{\partial \mathbf{p}_2}{\partial \rho_1} = \frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2} \frac{\partial \mathbf{p}_2}{\partial \mathbf{X}_2} \frac{\partial \mathbf{X}_2}{\partial \rho_1}$$

1.3. Initialization → Jacobian Derivation

derivation of inverse depth

마지막으로 Jacobian of inverse depth $\frac{\partial f(\mathbf{x})}{\partial \rho_1}$ 를 유도해야 한다.

$\frac{\partial f(\mathbf{x})}{\partial \rho_1}$ 는 다음과 같이 여러 연쇄적인 Jacobian의 곱으로 표현할 수 있다.

$$\frac{\partial f(\mathbf{x})}{\partial \rho_1} = \frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2} \frac{\partial \mathbf{p}_2}{\partial \rho_1} = \frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2} \frac{\partial \mathbf{p}_2}{\partial \mathbf{X}_2} \frac{\partial \mathbf{X}_2}{\partial \rho_1}$$

이 때, $\frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2} \frac{\partial \mathbf{p}_2}{\partial \mathbf{X}_2}$ 는 이전에 미리 구했으므로 $\frac{\partial \mathbf{X}_2}{\partial \rho_1}$ 만 추가적으로 구하면 된다.

$$\frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2} = \sqrt{w_h} \frac{\partial \mathbf{I}_2(\mathbf{p}_2)}{\partial \mathbf{p}_2} = \sqrt{w_h} [\nabla \mathbf{I}_x \quad \nabla \mathbf{I}_y]$$

$$\frac{\partial \mathbf{p}_2}{\partial \mathbf{X}_2} = \begin{bmatrix} \frac{\partial u_2}{\partial X_2} & \frac{\partial u_2}{\partial Y_2} & \frac{\partial u_2}{\partial Z_2} \\ \frac{\partial v_2}{\partial X_2} & \frac{\partial v_2}{\partial Y_2} & \frac{\partial v_2}{\partial Z_2} \end{bmatrix} = \begin{bmatrix} \rho_2 f_x & 0 & -\rho_2^2 f_x X_2 \\ 0 & \rho_2 f_y & -\rho_2^2 f_y Y_2 \end{bmatrix}$$

$$\frac{\partial \mathbf{X}_2}{\partial \rho_1} = ?$$

1.3. Initialization → Jacobian Derivation

derivation of inverse depth

$\frac{\partial \mathbf{p}_2}{\partial \rho_1} = \frac{\partial \mathbf{p}_2}{\partial \mathbf{X}_2} \frac{\partial \mathbf{X}_2}{\partial \rho_1}$ 에서 $\frac{\partial \mathbf{X}_2}{\partial \rho_1}$ 는 3차원 공간 상의 점 \mathbf{X}_2 과 Inverse depth ρ_1 사이의 관계를 나타내므로 이를 공식으로 표현하면 아래와 같다.

$$\mathbf{X}_2 = \mathbf{R}_{21}(\frac{\mathbf{K}^{-1}\mathbf{X}_1}{\rho_1}) + \mathbf{t}_{21}$$

\mathbf{X}_i	i번 카메라 좌표계에서 3차원 공간 상의 한 점
\mathbf{K}	2차원 이미지 평면 상의 한 점
\mathbf{R}_{21}	1번에서 2번 카메라로 변환하는 회전행렬 (3x3행렬)
\mathbf{t}_{21}	1번에서 2번 카메라로 변환하는 병진행렬 (3x1벡터)
ρ_1	1번 카메라 좌표계에서 한 점의 깊이 값의 역수 (inverse depth)

1.3. Initialization → Jacobian Derivation

derivation of inverse depth

\mathbf{X}_i	i번 카메라 좌표계에서 3차원 공간 상의 한 점
\mathbf{K}	2차원 이미지 평면 상의 한 점
\mathbf{R}_{21}	1번에서 2번 카메라로 변환하는 회전행렬 (3x3행렬)
\mathbf{t}_{21}	1번에서 2번 카메라로 변환하는 병진행렬 (3x1벡터)
ρ_1	1번 카메라 좌표계에서 한 점의 깊이 값의 역수 (inverse depth)

$\frac{\partial \mathbf{p}_2}{\partial \rho_1} = \frac{\partial \mathbf{p}_2}{\partial \mathbf{X}_2} \frac{\partial \mathbf{X}_2}{\partial \rho_1}$ 에서 $\frac{\partial \mathbf{X}_2}{\partial \rho_1}$ 는 3차원 공간 상의 점 \mathbf{X}_2 과 Inverse depth ρ_1 사이의 관계를 나타내므로 이를 공식으로 표현하면 아래와 같다.

$$\mathbf{X}_2 = \mathbf{R}_{21}\left(\frac{\mathbf{K}^{-1}\mathbf{X}_1}{\rho_1}\right) + \mathbf{t}_{21}$$

위 관계식을 사용하여 $\frac{\partial \mathbf{X}_2}{\partial \rho_1}$ 를 구해보면 아래와 같다.

$$\begin{aligned} \frac{\partial \mathbf{X}_2}{\partial \rho_1} &= \mathbf{R}_{21}\left(\frac{\mathbf{K}^{-1}\mathbf{X}_1}{\rho_1^2}\right) = -\rho_1^{-1} [X_2 - t_x \quad Y_2 - t_y \quad Z_2 - t_z]^T \\ &= -\rho_1^{-1} \rho_2 \begin{bmatrix} f_x(u'_2 t_z - t_x) \\ f_y(v'_2 t_z - t_y) \end{bmatrix} \end{aligned}$$

1.3. Initialization → Jacobian Derivation

derivation of inverse depth

\mathbf{X}_i	i번 카메라 좌표계에서 3차원 공간 상의 한 점
\mathbf{K}	2차원 이미지 평면 상의 한 점
\mathbf{R}_{21}	1번에서 2번 카메라로 변환하는 회전행렬 (3x3행렬)
\mathbf{t}_{21}	1번에서 2번 카메라로 변환하는 병진행렬 (3x1벡터)
ρ_1	1번 카메라 좌표계에서 한 점의 깊이 값의 역수 (inverse depth)

$\frac{\partial \mathbf{p}_2}{\partial \rho_1} = \frac{\partial \mathbf{p}_2}{\partial \mathbf{X}_2} \frac{\partial \mathbf{X}_2}{\partial \rho_1}$ 에서 $\frac{\partial \mathbf{X}_2}{\partial \rho_1}$ 는 3차원 공간 상의 점 \mathbf{X}_2 과 Inverse depth ρ_1 사이의 관계를 나타내므로 이를 공식으로 표현하면 아래와 같다.

$$\mathbf{X}_2 = \mathbf{R}_{21}\left(\frac{\mathbf{K}^{-1}\mathbf{X}_1}{\rho_1}\right) + \mathbf{t}_{21}$$

위 관계식을 사용하여 $\frac{\partial \mathbf{X}_2}{\partial \rho_1}$ 를 구해보면 아래와 같다.

$$\begin{aligned} \frac{\partial \mathbf{X}_2}{\partial \rho_1} &= \mathbf{R}_{21}\left(\frac{\mathbf{K}^{-1}\mathbf{X}_1}{\rho_1^2}\right) = -\rho_1^{-1} [X_2 - t_x \quad Y_2 - t_y \quad Z_2 - t_z]^T \\ &= -\rho_1^{-1} \rho_2 \begin{bmatrix} f_x(u'_2 t_z - t_x) \\ f_y(v'_2 t_z - t_y) \end{bmatrix} \end{aligned}$$

따라서 $\frac{\partial f(\mathbf{x})}{\partial \rho_1} = \frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2} \frac{\partial \mathbf{p}_2}{\partial \rho_1} = \frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_2} \frac{\partial \mathbf{p}_2}{\partial \mathbf{X}_2} \frac{\partial \mathbf{X}_2}{\partial \rho_1}$ 에서 $\frac{\partial f(\mathbf{x})}{\partial \rho_1}$ 는 다음과 같이 구할 수 있다.

$$\text{idepth(N)} \quad \therefore \frac{\partial f(\mathbf{x})}{\partial \rho_1} = \sqrt{w_h} \rho_1^{-1} \rho_2 (\nabla \mathbf{I}_x f_x(t_x - u'_2 t_z) + \nabla \mathbf{I}_y f_y(t_y - v'_2 t_z))$$

1.3. Initialization → Jacobian Derivation

Jacobians in DSO Initialization

정리해보면 Initialization에서 사용하는 자코비안들은 아래와 같다.

참고로 이 후 8개의 keyframe를 사용하여 **sliding window optimization**을 수행할 때는

camera intrinsics(f_x, f_y, c_x, c_y)가 최적화에 포함되므로 총 $12+N$ 개의 파라미터가 최적화 변수가 된다. (pose(6), photometric(2), intrinsics(4), idepth(N))

하지만 Initialization 과정에서는 intrinsics는 포함되지 않으므로 **8+N**개의 파라미터만 최적화에 사용된다.

idepth(N)

$$\frac{\partial f(\mathbf{x})}{\partial \rho_1} = \sqrt{w_h} \rho_1^{-1} \rho_2 (\nabla \mathbf{I}_x f_x(t_x - u'_2 t_z) + \nabla \mathbf{I}_y f_y(t_y - v'_2 t_z))$$

pose(6)

$$\frac{\partial f(\mathbf{x})}{\partial \delta \xi} = \sqrt{w_h} \begin{bmatrix} \nabla \mathbf{I}_x \rho_2 f_x \\ \nabla \mathbf{I}_y \rho_2 f_y \\ -\rho_2 (\nabla \mathbf{I}_x f_x u'_2 + \nabla \mathbf{I}_y f_y v'_2) \\ -\nabla \mathbf{I}_x f_x u'_2 v'_2 - \nabla \mathbf{I}_y f_y (1 + v'^2_2) \\ \nabla \mathbf{I}_x f_x (1 + u'^2_2) + \nabla \mathbf{I}_y f_y u'_2 v'_2 \\ -\nabla \mathbf{I}_x f_x v'_2 + \nabla \mathbf{I}_y f_y u'_2 \end{bmatrix}^T$$

photometric(2)

$$\begin{aligned} \frac{\partial f(\mathbf{x})}{\partial a} &= -\sqrt{w_h} \exp(a) \mathbf{I}_1(\mathbf{p}_1) \\ \frac{\partial f(\mathbf{x})}{\partial b} &= -\sqrt{w_h} \end{aligned}$$

1.3. Initialization → Jacobian Derivation

idepth(4), pose(6), photometric (2) 모두에 대한 Jacobian을 유도했으면 아래와 같이 합쳐준다.

$$\mathbf{J} = [\mathbf{J}_\rho \quad \mathbf{J}_y]_{1 \times (N+8)}^T$$

$$\mathbf{y} = [\delta\xi^T \quad a \quad b]$$

$$\mathbf{J}_\rho = \begin{bmatrix} \frac{\partial f(\mathbf{x})}{\partial \rho_1^{(1)}} & \dots & \frac{\partial f(\mathbf{x})}{\partial \rho_1^{(N)}} \end{bmatrix}_{1 \times N} \text{ for } \mathbf{x}_\rho$$

$$\mathbf{J}_y = \begin{bmatrix} \frac{\partial f(\mathbf{x})}{\partial \delta\xi} & \frac{\partial f(\mathbf{x})}{\partial a} & \frac{\partial f(\mathbf{x})}{\partial b} \end{bmatrix}_{1 \times 8} \text{ for } \mathbf{x}_y$$

1.3. Initialization → Jacobian Derivation

idepth(4), pose(6), photometric (2) 모두에 대한 Jacobian을 유도했으면 아래와 같이 합쳐준다.

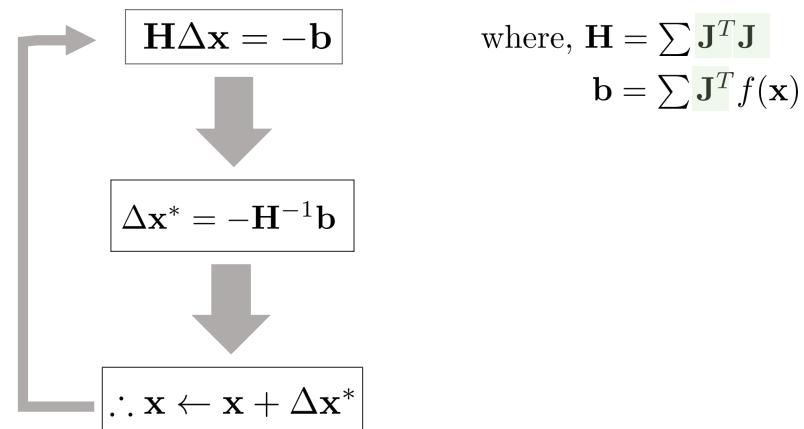
$$\mathbf{J} = [\mathbf{J}_\rho \quad \mathbf{J}_y]^T_{1 \times (N+8)}$$

$$\mathbf{y} = [\delta\xi^T \quad a \quad b]$$

$$\mathbf{J}_\rho = \left[\frac{\partial f(\mathbf{x})}{\partial \rho_1^{(1)}} \quad \dots \quad \frac{\partial f(\mathbf{x})}{\partial \rho_1^{(N)}} \right]_{1 \times N} \text{ for } \mathbf{x}_\rho$$

$$\mathbf{J}_y = \left[\frac{\partial f(\mathbf{x})}{\partial \delta\xi} \quad \frac{\partial f(\mathbf{x})}{\partial a} \quad \frac{\partial f(\mathbf{x})}{\partial b} \right]_{1 \times 8} \text{ for } \mathbf{x}_y$$

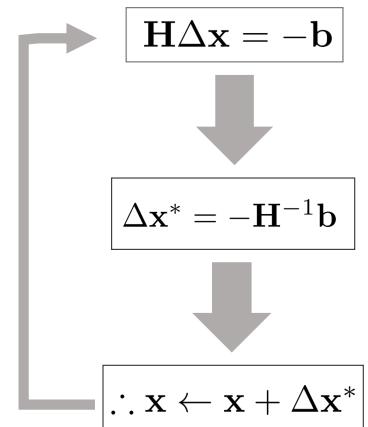
최종적으로 위 Jacobian \mathbf{J} 를 사용하여 state variable $\mathbf{x} = [\rho_1^{(1)}, \dots, \rho_1^{(N)}, \delta\xi^T, a, b]^T_{(N+8) \times 1} = [\mathbf{x}_\rho \quad \mathbf{x}_y]^T$ 를 업데이트할 수 있다.



where, $\mathbf{H} = \sum \mathbf{J}^T \mathbf{J}$
 $\mathbf{b} = \sum \mathbf{J}^T f(\mathbf{x})$

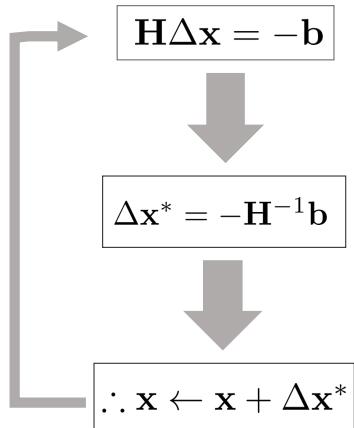
1.4. Initialization → Solving the incremental equation (Schur Complement)

Jacobian까지 모두 유도를 마쳤으면 Gauss Newton method를 사용해서 아래와 같은 최적화를 반복해서 풀어야 한다.



1.4. Initialization → Solving the incremental equation (Schur Complement)

Jacobian까지 모두 유도를 마쳤으면 Gauss Newton method를 사용해서 아래와 같은 최적화를 반복해서 풀어야 한다.



하지만 실제 위 최적화 공식을 반복적으로 풀면 매우 느린 속도로 인해 정상적으로 Pose tracking을 할 수 없다.

이는 \mathbf{H}^{-1} 를 계산할 때 매우 큰 \mathbf{H} 행렬의 특성 상 역함수를 구하는데 매우 많은 시간이 소모되기 때문이다.

해당 공식을 조금 더 자세히 표현해보면 다음과 같다.

$$\mathbf{H}\Delta\mathbf{x} = \mathbf{b} \quad \xrightarrow{\text{(↑부호를 } \mathbf{b} \text{ 안으로 삽입)}} \quad \begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho\mathbf{y}} \\ \mathbf{H}_{\mathbf{y}\rho} & \mathbf{H}_{\mathbf{y}\mathbf{y}} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{x}_\rho \\ \Delta\mathbf{x}_\mathbf{y} \end{bmatrix} = \begin{bmatrix} \mathbf{b}_\rho \\ \mathbf{b}_\mathbf{y} \end{bmatrix}$$

$\mathbf{y} = [\delta\xi^T \ a \ b]$
$\mathbf{H}_{\rho\rho} = \sum \mathbf{J}_\rho^T \mathbf{J}_\rho$
$\mathbf{H}_{\rho\mathbf{y}} = \sum \mathbf{J}_\rho^T \mathbf{J}_\mathbf{y}$
$\mathbf{H}_{\mathbf{y}\rho} = \sum \mathbf{J}_\mathbf{y}^T \mathbf{J}_\rho$
$\mathbf{H}_{\mathbf{y}\mathbf{y}} = \sum \mathbf{J}_\mathbf{y}^T \mathbf{J}_\mathbf{y}$
$\mathbf{b}_\rho = -\sum \mathbf{J}_\rho^T f(\mathbf{x})$
$\mathbf{b}_\mathbf{y} = -\sum \mathbf{J}_\mathbf{y}^T f(\mathbf{x})$

1.4. Initialization → Solving the incremental equation (Schur Complement)

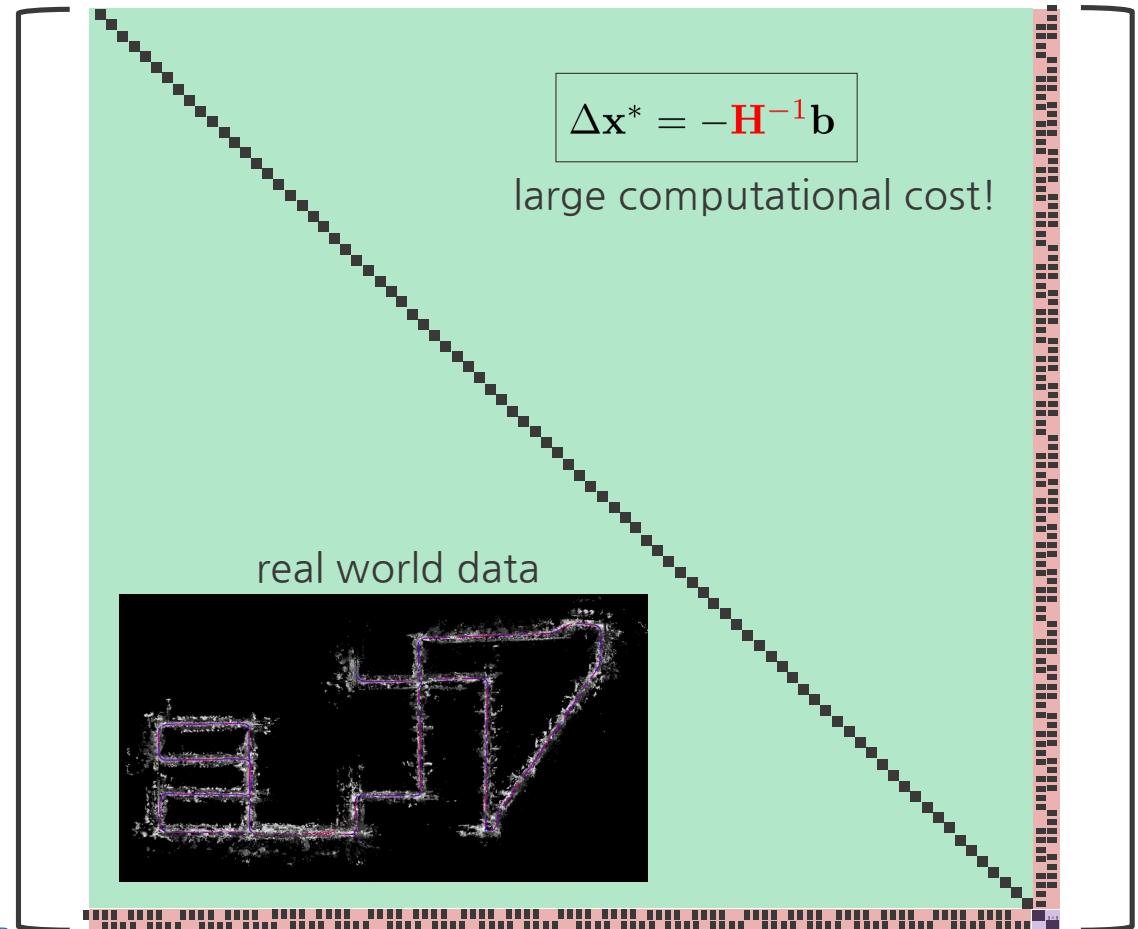
Hessian Matrix \mathbf{H} 를 전개하면 내부 구조는 다음과 같다

1.4. Initialization → Solving the incremental equation (Schur Complement)

일반적인 경우 3차원 점의 개수가 카메라 pose보다 훨씬 많으므로 다음과 같은 거대한 matrix가 생성된다.

따라서 역행렬을 계산할 때 매우 큰 연산시간이 소요되므로 이를 빠르게 계산할 수 있는 다른 방법이 필요하다

$$\mathbf{H} = \mathbf{J}^T \mathbf{J} = \begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho\mathbf{y}} \\ \mathbf{H}_{\mathbf{y}\rho} & \mathbf{H}_{\mathbf{y}\mathbf{y}} \end{bmatrix} =$$



1.4. Initialization → Solving the incremental equation (Schur Complement)

이 때, Schur Complement를 사용하면 computational cost를 대폭 낮출 수 있다

$$\begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho\mathbf{y}} \\ \mathbf{H}_{\mathbf{y}\rho} & \mathbf{H}_{\mathbf{y}\mathbf{y}} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x}_\rho \\ \Delta \mathbf{x}_\mathbf{y} \end{bmatrix} = \begin{bmatrix} \mathbf{b}_\rho \\ \mathbf{b}_\mathbf{y} \end{bmatrix}$$

$\mathbf{y} = [\delta\xi^T \ a \ b]$
$\mathbf{H}_{\rho\rho} = \sum \mathbf{J}_\rho^T \mathbf{J}_\rho$
$\mathbf{H}_{\rho\mathbf{y}} = \sum \mathbf{J}_\rho^T \mathbf{J}_\mathbf{y}$
$\mathbf{H}_{\mathbf{y}\rho} = \sum \mathbf{J}_\mathbf{y}^T \mathbf{J}_\rho$
$\mathbf{H}_{\mathbf{y}\mathbf{y}} = \sum \mathbf{J}_\mathbf{y}^T \mathbf{J}_\mathbf{y}$
$\mathbf{b}_\rho = - \sum \mathbf{J}_\rho^T f(\mathbf{x})$
$\mathbf{b}_\mathbf{y} = - \sum \mathbf{J}_\mathbf{y}^T f(\mathbf{x})$
$\mathbf{H}_{sc} = \mathbf{H}_{\mathbf{y}\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{H}_{\rho\mathbf{y}}$
$\mathbf{b}_{sc} = \mathbf{H}_{\mathbf{y}\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{b}_\rho$

1.4. Initialization → Solving the incremental equation (Schur Complement)

이 때, Schur Complement를 사용하면 computational cost를 대폭 낮출 수 있다

$$\begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho y} \\ \mathbf{H}_{y\rho} & \mathbf{H}_{yy} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x}_\rho \\ \Delta \mathbf{x}_y \end{bmatrix} = \begin{bmatrix} \mathbf{b}_\rho \\ \mathbf{b}_y \end{bmatrix}$$

이를 위해 양변에 아래와 같은 특수한 형태의 행렬을 양변에 곱해준다.



$$\begin{bmatrix} \mathbf{I} & \mathbf{0} \\ -\mathbf{H}_{y\rho}\mathbf{H}_{\rho\rho}^{-1} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho y} \\ \mathbf{H}_{y\rho} & \mathbf{H}_{yy} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x}_\rho \\ \Delta \mathbf{x}_y \end{bmatrix} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ -\mathbf{H}_{y\rho}\mathbf{H}_{\rho\rho}^{-1} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{b}_\rho \\ \mathbf{b}_y \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho y} \\ \mathbf{0} & -\mathbf{H}_{y\rho}\mathbf{H}_{\rho\rho}^{-1}\mathbf{H}_{\rho y} + \mathbf{H}_{yy} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x}_\rho \\ \Delta \mathbf{x}_y \end{bmatrix} = \begin{bmatrix} \mathbf{b}_\rho \\ -\mathbf{H}_{y\rho}\mathbf{H}_{\rho\rho}^{-1}\mathbf{b}_\rho + \mathbf{b}_y \end{bmatrix}$$

$\mathbf{y} = [\delta\xi^T \ a \ b]$
$\mathbf{H}_{\rho\rho} = \sum \mathbf{J}_\rho^T \mathbf{J}_\rho$
$\mathbf{H}_{\rho y} = \sum \mathbf{J}_\rho^T \mathbf{J}_y$
$\mathbf{H}_{y\rho} = \sum \mathbf{J}_y^T \mathbf{J}_\rho$
$\mathbf{H}_{yy} = \sum \mathbf{J}_y^T \mathbf{J}_y$
$\mathbf{b}_\rho = -\sum \mathbf{J}_\rho^T f(\mathbf{x})$
$\mathbf{b}_y = -\sum \mathbf{J}_y^T f(\mathbf{x})$
$\mathbf{H}_{sc} = \mathbf{H}_{y\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{H}_{\rho y}$
$\mathbf{b}_{sc} = \mathbf{H}_{y\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{b}_\rho$

1.4. Initialization → Solving the incremental equation (Schur Complement)

이 때, Schur Complement를 사용하면 computational cost를 대폭 낮출 수 있다

$$\begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho y} \\ \mathbf{H}_{y\rho} & \mathbf{H}_{yy} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x}_\rho \\ \Delta \mathbf{x}_y \end{bmatrix} = \begin{bmatrix} \mathbf{b}_\rho \\ \mathbf{b}_y \end{bmatrix}$$

이를 위해 양변에 아래와 같은 특수한 형태의 행렬을 양변에 곱해준다.

$\mathbf{y} = [\delta\xi^T \ a \ b]$
$\mathbf{H}_{\rho\rho} = \sum \mathbf{J}_\rho^T \mathbf{J}_\rho$
$\mathbf{H}_{\rho y} = \sum \mathbf{J}_\rho^T \mathbf{J}_y$
$\mathbf{H}_{y\rho} = \sum \mathbf{J}_y^T \mathbf{J}_\rho$
$\mathbf{H}_{yy} = \sum \mathbf{J}_y^T \mathbf{J}_y$
$\mathbf{b}_\rho = -\sum \mathbf{J}_\rho^T f(\mathbf{x})$
$\mathbf{b}_y = -\sum \mathbf{J}_y^T f(\mathbf{x})$
$\mathbf{H}_{sc} = \mathbf{H}_{y\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{H}_{\rho y}$
$\mathbf{b}_{sc} = \mathbf{H}_{y\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{b}_\rho$

$$\begin{bmatrix} \mathbf{I} & \mathbf{0} \\ -\mathbf{H}_{y\rho} \mathbf{H}_{\rho\rho}^{-1} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho y} \\ \mathbf{H}_{y\rho} & \mathbf{H}_{yy} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x}_\rho \\ \Delta \mathbf{x}_y \end{bmatrix} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ -\mathbf{H}_{y\rho} \mathbf{H}_{\rho\rho}^{-1} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{b}_\rho \\ \mathbf{b}_y \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho y} \\ \mathbf{0} & -\mathbf{H}_{y\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{H}_{\rho y} + \mathbf{H}_{yy} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x}_\rho \\ \Delta \mathbf{x}_y \end{bmatrix} = \begin{bmatrix} \mathbf{b}_\rho \\ -\mathbf{H}_{y\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{b}_\rho + \mathbf{b}_y \end{bmatrix}$$

위 식을 전개해서 $\Delta \mathbf{x}_y$ $\Delta \mathbf{x}_\rho$ 의 값을 순차적으로 계산할 수 있다. 위 식을 전개하면 $\Delta \mathbf{x}_y$ 만 존재하는 항이 유도된다.

1 $\Delta \mathbf{x}_y = (\mathbf{H}_{yy} - \mathbf{H}_{sc})^{-1} (\mathbf{b}_y - \mathbf{b}_{sc})$

1.4. Initialization → Solving the incremental equation (Schur Complement)

이 때, Schur Complement를 사용하면 computational cost를 대폭 낮출 수 있다

$$\begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho y} \\ \mathbf{H}_{y\rho} & \mathbf{H}_{yy} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x}_\rho \\ \Delta \mathbf{x}_y \end{bmatrix} = \begin{bmatrix} \mathbf{b}_\rho \\ \mathbf{b}_y \end{bmatrix}$$

이를 위해 양변에 아래와 같은 특수한 형태의 행렬을 양변에 곱해준다.

$$\begin{aligned} \mathbf{y} &= [\delta \xi^T \ a \ b] \\ \mathbf{H}_{\rho\rho} &= \sum \mathbf{J}_\rho^T \mathbf{J}_\rho \\ \mathbf{H}_{\rho y} &= \sum \mathbf{J}_\rho^T \mathbf{J}_y \\ \mathbf{H}_{y\rho} &= \sum \mathbf{J}_y^T \mathbf{J}_\rho \\ \mathbf{H}_{yy} &= \sum \mathbf{J}_y^T \mathbf{J}_y \\ \mathbf{b}_\rho &= -\sum \mathbf{J}_\rho^T f(\mathbf{x}) \\ \mathbf{b}_y &= -\sum \mathbf{J}_y^T f(\mathbf{x}) \\ \mathbf{H}_{sc} &= \mathbf{H}_{y\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{H}_{\rho y} \\ \mathbf{b}_{sc} &= \mathbf{H}_{y\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{b}_\rho \end{aligned}$$

$$\begin{bmatrix} \mathbf{I} & 0 \\ -\mathbf{H}_{y\rho} \mathbf{H}_{\rho\rho}^{-1} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho y} \\ \mathbf{H}_{y\rho} & \mathbf{H}_{yy} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x}_\rho \\ \Delta \mathbf{x}_y \end{bmatrix} = \begin{bmatrix} \mathbf{I} & 0 \\ -\mathbf{H}_{y\rho} \mathbf{H}_{\rho\rho}^{-1} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{b}_\rho \\ \mathbf{b}_y \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho y} \\ 0 & -\mathbf{H}_{y\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{H}_{\rho y} + \mathbf{H}_{yy} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x}_\rho \\ \Delta \mathbf{x}_y \end{bmatrix} = \begin{bmatrix} \mathbf{b}_\rho \\ -\mathbf{H}_{y\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{b}_\rho + \mathbf{b}_y \end{bmatrix}$$

위 식을 전개해서 $\Delta \mathbf{x}_y$ $\Delta \mathbf{x}_\rho$ 의 값을 순차적으로 계산할 수 있다. 위 식을 전개하면 $\Delta \mathbf{x}_y$ 만 존재하는 항이 유도된다.

$$① \quad \Delta \mathbf{x}_y = (\mathbf{H}_{yy} - \mathbf{H}_{sc})^{-1} (\mathbf{b}_y - \mathbf{b}_{sc})$$

위 식을 통해 $\Delta \mathbf{x}_y$ 를 먼저 계산한 후 이를 토대로 $\Delta \mathbf{x}_\rho$ 를 계산한다. 이를 통해 역행렬을 구하는 것보다 더욱 빠르게 최적화 공식을 풀 수 있다.

$$② \quad \Delta \mathbf{x}_\rho = \mathbf{H}_{\rho\rho}^{-1} (\mathbf{b}_\rho - \mathbf{H}_{\rho y} \Delta \mathbf{x}_y)$$

1.4. Initialization → Solving the incremental equation (Schur Complement)

Inverse depth와 관련 있는 항인 $\mathbf{H}_{\rho\rho}$ 는 $N \times N$ 크기의 대각행렬(diagonal matrix)이므로 쉽게 역행렬을 구할 수 있다.

따라서 $\mathbf{H}_{sc}, \mathbf{b}_{sc}$ 또한 다음과 같이 비교적 간단하게 계산할 수 있다.

$$\mathbf{H}_{sc} = \mathbf{H}_{y\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{H}_{\rho y} = \sum \frac{1}{\mathbf{J}_\rho \mathbf{J}_\rho^T} (\mathbf{J}_\rho^T \mathbf{J}_y)^T \mathbf{J}_\rho^T \mathbf{J}_y$$

$$\mathbf{b}_{sc} = \mathbf{H}_{y\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{b}_\rho = - \sum \frac{1}{\mathbf{J}_\rho \mathbf{J}_\rho^T} (\mathbf{J}_\rho^T \mathbf{J}_y)^T \mathbf{J}_\rho^T f(\mathbf{x})$$

$$\begin{aligned}\mathbf{y} &= [\delta\xi^T \ a \ b] \\ \mathbf{H}_{\rho\rho} &= \sum \mathbf{J}_\rho^T \mathbf{J}_\rho \\ \mathbf{H}_{\rho y} &= \sum \mathbf{J}_\rho^T \mathbf{J}_y \\ \mathbf{H}_{y\rho} &= \sum \mathbf{J}_y^T \mathbf{J}_\rho \\ \mathbf{H}_{yy} &= \sum \mathbf{J}_y^T \mathbf{J}_y \\ \mathbf{b}_\rho &= - \sum \mathbf{J}_\rho^T f(\mathbf{x}) \\ \mathbf{b}_y &= - \sum \mathbf{J}_y^T f(\mathbf{x}) \\ \mathbf{H}_{sc} &= \mathbf{H}_{y\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{H}_{\rho y} \\ \mathbf{b}_{sc} &= \mathbf{H}_{y\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{b}_\rho\end{aligned}$$

1.4. Initialization → Solving the incremental equation (Schur Complement)

Inverse depth와 관련 있는 항인 $\mathbf{H}_{\rho\rho}$ 는 $N \times N$ 크기의 대각행렬(diagonal matrix)이므로 쉽게 역행렬을 구할 수 있다.

따라서 $\mathbf{H}_{sc}, \mathbf{b}_{sc}$ 또한 다음과 같이 비교적 간단하게 계산할 수 있다.

$$\mathbf{H}_{sc} = \mathbf{H}_{y\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{H}_{\rho y} = \sum \frac{1}{\mathbf{J}_\rho \mathbf{J}_\rho^T} (\mathbf{J}_\rho^T \mathbf{J}_y)^T \mathbf{J}_\rho^T \mathbf{J}_y$$

$$\mathbf{b}_{sc} = \mathbf{H}_{y\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{b}_\rho = - \sum \frac{1}{\mathbf{J}_\rho \mathbf{J}_\rho^T} (\mathbf{J}_\rho^T \mathbf{J}_y)^T \mathbf{J}_\rho^T f(\mathbf{x})$$

$$\begin{aligned}\mathbf{y} &= [\delta\xi^T \ a \ b] \\ \mathbf{H}_{\rho\rho} &= \sum \mathbf{J}_\rho^T \mathbf{J}_\rho \\ \mathbf{H}_{\rho y} &= \sum \mathbf{J}_\rho^T \mathbf{J}_y \\ \mathbf{H}_{y\rho} &= \sum \mathbf{J}_y^T \mathbf{J}_\rho \\ \mathbf{H}_{yy} &= \sum \mathbf{J}_y^T \mathbf{J}_y \\ \mathbf{b}_\rho &= - \sum \mathbf{J}_\rho^T f(\mathbf{x}) \\ \mathbf{b}_y &= - \sum \mathbf{J}_y^T f(\mathbf{x}) \\ \mathbf{H}_{sc} &= \mathbf{H}_{y\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{H}_{\rho y} \\ \mathbf{b}_{sc} &= \mathbf{H}_{y\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{b}_\rho\end{aligned}$$

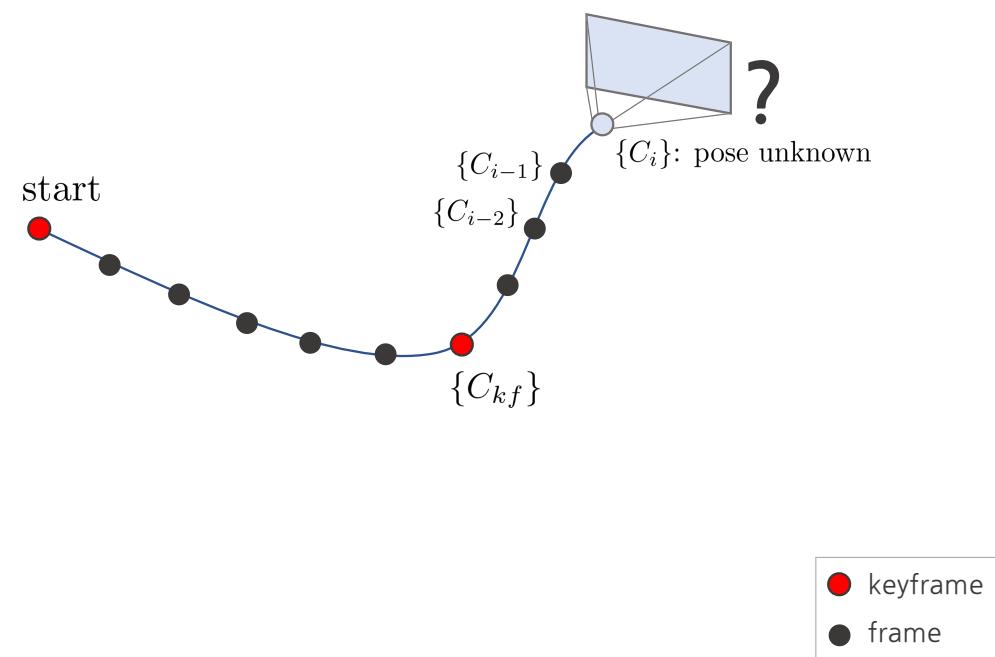
최종적으로 damping 계수 λ 기반의 Levenberg-Marquardt 방법을 사용하여 Optimization을 수행한다.

$$(\mathbf{H} + \lambda \mathbf{I}) \Delta \mathbf{x} = \mathbf{b}$$

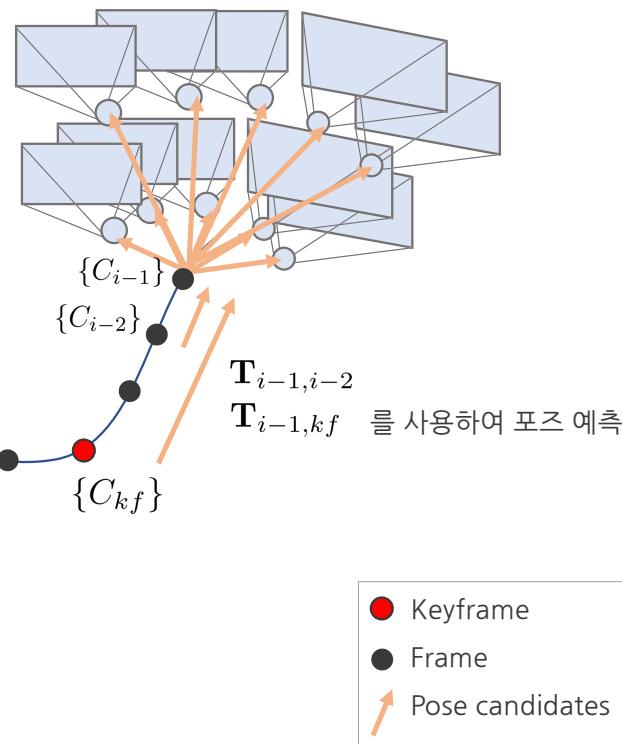
2. Frames

2.1. Frames → Pose Tracking

새로운 이미지가 들어오면 해당 이미지에 해당하는 3차원 자세를 아직 모르기 때문에 이를 예측하기 위해 DSO는 가장 최근 두 프레임 $\{C_{i-1,2}\}$ 과 가장 근접한 키프레임 $\{C_{kf}\}$ 의 정보를 사용한다.



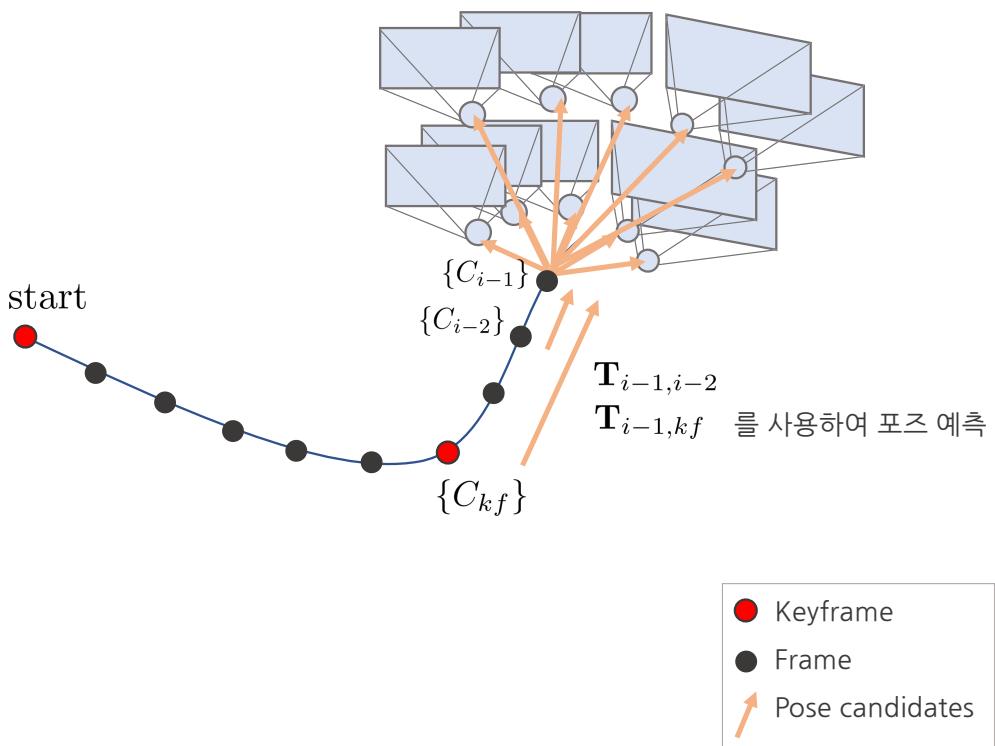
2.1. Frames → Pose Tracking



새로운 이미지가 들어오면 해당 이미지에 해당하는 3차원 자세를 아직 모르기 때문에 이를 예측하기 위해 DSO는 가장 최근 두 프레임 $\{C_{i-1,2}\}$ 과 가장 근접한 키프레임 $\{C_{kf}\}$ 의 정보를 사용한다.

두 프레임과 한 개의 키프레임을 바탕으로 이들의 상대 포즈를 구한 후 $T_{i-1,i-2}$ $T_{i-1,kf}$ 정지, 직진, 후진, 회전 등 다양한 케이스를 예측한다.

2.1. Frames → Pose Tracking

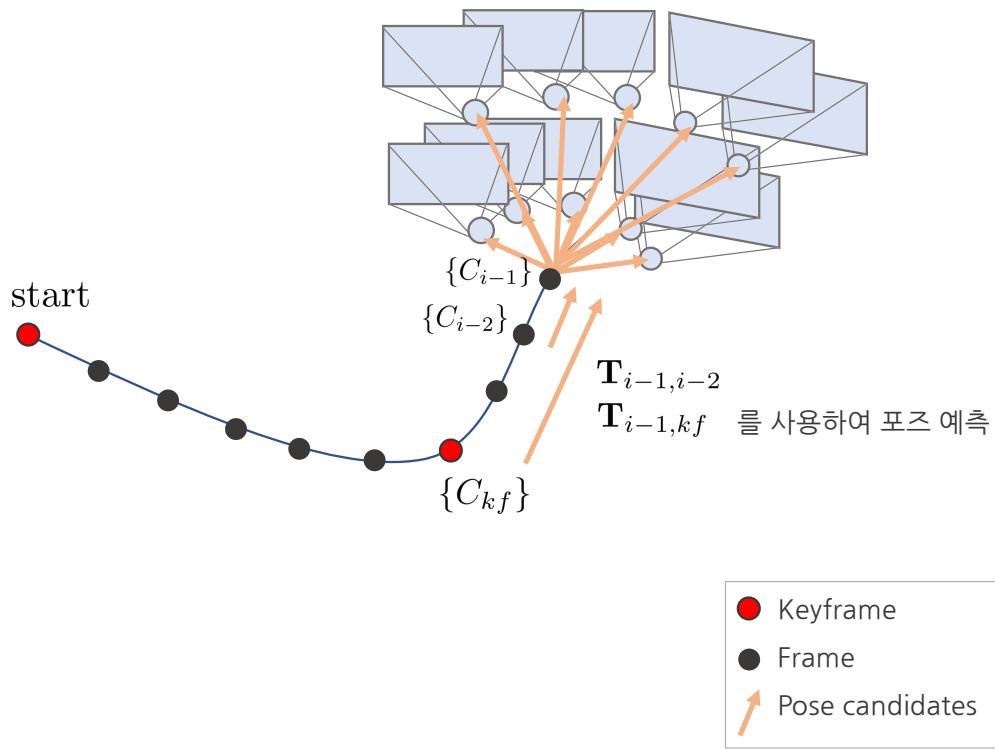


새로운 이미지가 들어오면 해당 이미지에 해당하는 3차원 자세를 아직 모르기 때문에 이를 예측하기 위해 DSO는 가장 최근 두 프레임 $\{C_{i-1,2}\}$ 과 가장 근접한 키프레임 $\{C_{kf}\}$ 의 정보를 사용한다.

두 프레임과 한 개의 키프레임을 바탕으로 이들의 상대 포즈를 구한 후 $T_{i-1,i-2}$ $T_{i-1,kf}$ 정지, 직진, 후진, 회전 등 다양한 케이스를 예측한다.

이 때 가장 높은 이미지 피라미드(가장 작은)부터 낮은 이미지 피라미드(원본)까지 순차적으로 사용하는 **coarse-to-fine** 방법을 통해 현재 프레임의 3차원 자세를 예측한다.

2.1. Frames → Pose Tracking



새로운 이미지가 들어오면 해당 이미지에 해당하는 3차원 자세를 아직 모르기 때문에 이를 예측하기 위해 DSO는 가장 최근 두 프레임 $\{C_{i-1,2}\}$ 과 가장 근접한 키프레임 $\{C_{kf}\}$ 의 정보를 사용한다.

두 프레임과 한 개의 키프레임을 바탕으로 이들의 상대 포즈를 구한 후 $T_{i-1,i-2}$ $T_{i-1,kf}$ 정지, 직진, 후진, 회전 등 다양한 케이스를 예측한다.

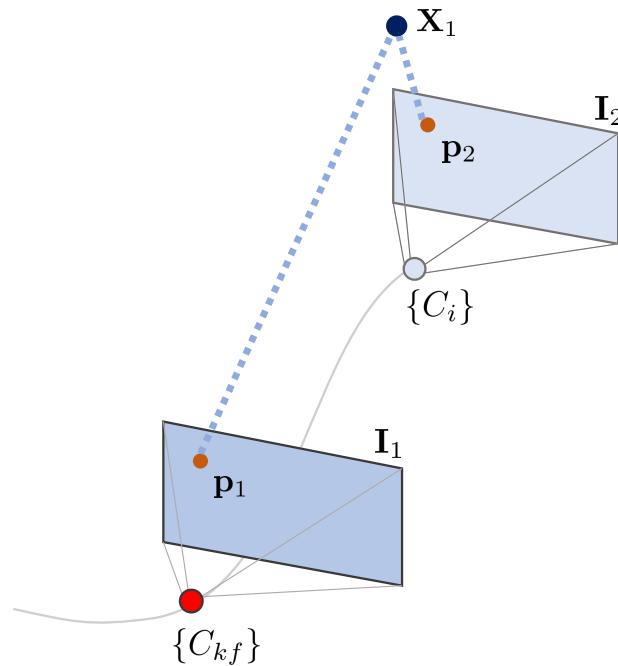
이 때 가장 높은 이미지 피라미드(가장 작은)부터 낮은 이미지 피라미드(원본)까지 순차적으로 사용하는 **coarse-to-fine** 방법을 통해 현재 프레임의 3차원 자세를 예측한다.

각 포즈 후보마다 순차적으로 밝기 오차(Photometric Error)를 계산하고 이를 최적화하여 최적의 포즈를 구한다. 그리고 최적의 포즈에서 다시 한 번 밝기 오차를 구한 후(resNew) 이전 오차(resOld)와 비교하여 충분히 오차가 감소하였는지 판단한다. 충분히 감소하였다면 다른 포즈 후보를 고려하지 않고 루프를 탈출한다. (coarseTracker::trackNewestCoarse() 함수 참조)

2.1. Frames → Pose Tracking

여러 후보 포즈들 중 하나의 포즈를 선택한 후 밝기 오차(Photometric Error)를 계산한다.

$$\mathbf{r} = \mathbf{I}_2(\mathbf{p}_2) - \exp(a)(\mathbf{I}_1(\mathbf{p}_1) - b_0) - b \quad \text{where, } \mathbf{p}_2 = \pi(\exp(\xi_{21}^\wedge)\mathbf{X}_1) = \pi(\exp(\xi_{21}^\wedge)\pi^{-1}(\mathbf{p}_1))$$



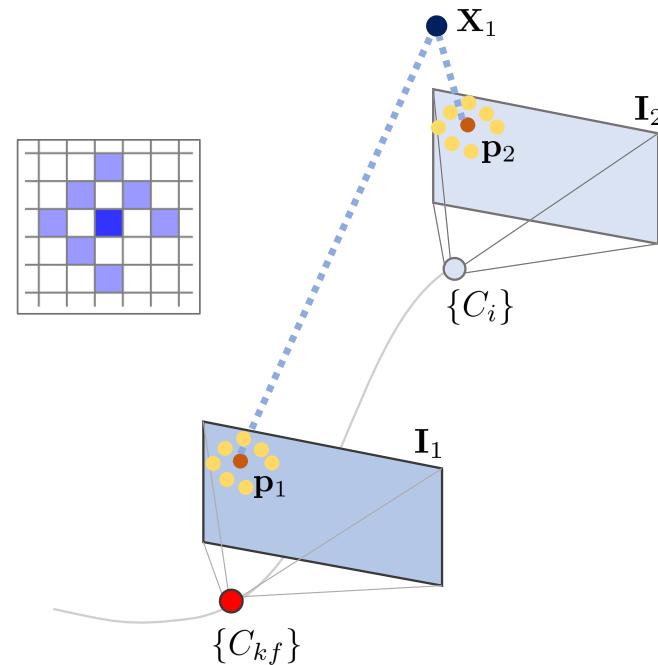
$\mathbf{I}_i(\mathbf{p}_j)$	i번째 이미지에서 j번째 점의 밝기 (grayscale)(intensity)(0~255단위)
t_i	i번째 이미지가 촬영된 순간의 exposure time (또는 Shutter time)(ms단위)
a, b	두 이미지 간曝光 time을 고려하여 이미지 밝기를 조절하는 파라미터

2.1. Frames → Pose Tracking

여러 후보 포즈들 중 하나의 포즈를 선택한 후 밝기 오차(Photometric Error)를 계산한다.

이 때, 한 점의 광도 오차만 계산하는 것이 아닌 주변 8개의 점들의 밝기 오차를 모두 계산한다.

$$\mathbf{r} = \mathbf{I}_2(\mathbf{p}_2) - \exp(a)(\mathbf{I}_1(\mathbf{p}_1) - b_0) - b \quad \text{where, } \mathbf{p}_2 = \pi(\exp(\xi_{21}^\wedge)\mathbf{X}_1) = \pi(\exp(\xi_{21}^\wedge)\pi^{-1}(\mathbf{p}_1))$$



$\mathbf{I}_i(\mathbf{p}_j)$	i번째 이미지에서 j번째 점의 밝기 (grayscale)(intensity)(0~255단위)
t_i	i번째 이미지가 촬영된 순간의 exposure time (또는 Shutter time)(ms단위)
a, b	두 이미지 간曝光时间을 고려하여 이미지 밝기를 조절하는 파라미터

2.1. Frames → Pose Tracking

여러 후보 포즈들 중 하나의 포즈를 선택한 후 밝기 오차(Photometric Error)를 계산한다.

이 때, 한 점의 광도 오차만 계산하는 것이 아닌 주변 8개의 점들의 밝기 오차를 모두 계산한다.

$$\mathbf{r} = \mathbf{I}_2(\mathbf{p}_2) - \exp(a)(\mathbf{I}_1(\mathbf{p}_1) - b_0) - b \quad \text{where, } \mathbf{p}_2 = \pi(\exp(\xi_{21}^\wedge)\mathbf{X}_1) = \pi(\exp(\xi_{21}^\wedge)\pi^{-1}(\mathbf{p}_1))$$

다음으로 해당 비선형 함수를 최적화하기 위해 Jacobian을 계산한다.

해당 Jacobian은 이미 Initialization 단계에서 유도 했으므로 유도 과정은 생략한다.

이 때, $f(\mathbf{x}) = \sqrt{w_h} \mathbf{r}$ 에서 \mathbf{r} 만 우선 고려한다.

pose(6)

photometric(2)

$$\frac{\partial \mathbf{r}}{\partial a} = \exp(a)\mathbf{I}_1(b_0 - \mathbf{p}_1)$$

$$\frac{\partial \mathbf{r}}{\partial b} = -1$$

$$\frac{\partial \mathbf{r}}{\partial a} = -\exp(a)\mathbf{I}_1(\mathbf{p}_1)$$

$$\frac{\partial \mathbf{r}}{\partial \delta \xi} = \begin{bmatrix} \nabla \mathbf{I}_x \rho_2 f_x \\ \nabla \mathbf{I}_y \rho_2 f_y \\ -\rho_2 (\nabla \mathbf{I}_x f_x u'_2 + \nabla \mathbf{I}_y f_y v'_2) \\ -\nabla \mathbf{I}_x f_x u'_2 v'_2 - \nabla \mathbf{I}_y f_y (1 + v'^2_2) \\ \nabla \mathbf{I}_x f_x (1 + u'^2_2) + \nabla \mathbf{I}_y f_y u'_2 v'_2 \\ -\nabla \mathbf{I}_x f_x v'_2 + \nabla \mathbf{I}_y f_y u'_2 \end{bmatrix}^T$$



Initialization 단계와 다르게 초기값 b_0 추가됨

$$\frac{\partial \mathbf{r}}{\partial a} = \exp(a)\mathbf{I}_1(b_0 - \mathbf{p}_1)$$

2.1. Frames → Pose Tracking

여러 후보 포즈들 중 하나의 포즈를 선택한 후 밝기 오차(Photometric Error)를 계산한다.

이 때, 한 점의 광도 오차만 계산하는 것이 아닌 주변 8개의 점들의 밝기 오차를 모두 계산한다.

$$\mathbf{r} = \mathbf{I}_2(\mathbf{p}_2) - \exp(a)(\mathbf{I}_1(\mathbf{p}_1) - b_0) - b \quad \text{where, } \mathbf{p}_2 = \pi(\exp(\xi_{21}^\wedge)\mathbf{X}_1) = \pi(\exp(\xi_{21}^\wedge)\pi^{-1}(\mathbf{p}_1))$$

다음으로 해당 비선형 함수를 최적화하기 위해 Jacobian을 계산한다.

해당 Jacobian은 이미 Initialization 단계에서 유도 했으므로 유도 과정은 생략한다.

이 때, $f(\mathbf{x}) = \sqrt{w_h} \mathbf{r}$ 에서 \mathbf{r} 만 우선 고려한다.

pose(6)

photometric(2)

$$\begin{aligned}\frac{\partial \mathbf{r}}{\partial a} &= \exp(a)\mathbf{I}_1(b_0 - \mathbf{p}_1) \\ \frac{\partial \mathbf{r}}{\partial b} &= -1\end{aligned}$$

$$\frac{\partial \mathbf{r}}{\partial \delta \xi} = \begin{bmatrix} \nabla \mathbf{I}_x \rho_2 f_x \\ \nabla \mathbf{I}_y \rho_2 f_y \\ -\rho_2 (\nabla \mathbf{I}_x f_x u'_2 + \nabla \mathbf{I}_y f_y v'_2) \\ -\nabla \mathbf{I}_x f_x u'_2 v'_2 - \nabla \mathbf{I}_y f_y (1 + v'^2_2) \\ \nabla \mathbf{I}_x f_x (1 + u'^2_2) + \nabla \mathbf{I}_y f_y u'_2 v'_2 \\ -\nabla \mathbf{I}_x f_x v'_2 + \nabla \mathbf{I}_y f_y u'_2 \end{bmatrix}^T$$

해당 Jacobian을 사용하여 \mathbf{H}, \mathbf{b} 를 각각 계산한다.

$$\begin{aligned}\mathbf{H} &= \sum \mathbf{J}^T \mathbf{J} \\ \mathbf{b} &= \sum \mathbf{J}^T \mathbf{r}\end{aligned}$$

where, $\mathbf{J} = \left[\frac{\partial f(\mathbf{x})}{\partial \delta \xi} \quad \frac{\partial f(\mathbf{x})}{\partial a} \quad \frac{\partial f(\mathbf{x})}{\partial b} \right]_{8 \times 1}$

2.1. Frames → Pose Tracking

\mathbf{H}, \mathbf{b} 를 각각 계산한 다음 Levenberg-Marquardt 방법을 통해 반복적으로 에러를 감소시킨다. 초기 $\lambda = 0.01$ 로 설정한다.

$$(\mathbf{H} + \lambda \mathbf{I})\Delta \mathbf{x} = \mathbf{b}$$

$$\mathbf{x} \leftarrow \mathbf{x} + \Delta \mathbf{x} \quad \text{where, } \Delta \mathbf{x} = [\delta \xi^T, \delta a, \delta b]_{1 \times 8}^T$$

2.1. Frames → Pose Tracking

\mathbf{H}, \mathbf{b} 를 각각 계산한 다음 Levenberg-Marquardt 방법을 통해 반복적으로 에러를 감소시킨다. 초기 $\lambda = 0.01$ 로 설정한다.

$$(\mathbf{H} + \lambda \mathbf{I})\Delta \mathbf{x} = \mathbf{b}$$

$$\mathbf{x} \leftarrow \mathbf{x} + \Delta \mathbf{x} \quad \text{where, } \Delta \mathbf{x} = [\delta \xi^T, \delta a, \delta b]_{1 \times 8}^T$$

이 때 $\mathbf{x}_{1:6} \in \text{SE}(3)$ 는 카메라의 포즈를 의미하므로 증분량 $\Delta \mathbf{x}_{1:6}^\wedge \in \text{se}(3)$ 는 실제로 아래와 같이 업데이트 된다.

$$\mathbf{x}_{1:6} \leftarrow \exp(\Delta \mathbf{x}_{1:6}^\wedge) \mathbf{x}_{1:6}$$

2.1. Frames → Pose Tracking

\mathbf{H}, \mathbf{b} 를 각각 계산한 다음 Levenberg-Marquardt 방법을 통해 반복적으로 에러를 감소시킨다. 초기 $\lambda = 0.01$ 로 설정한다.

$$(\mathbf{H} + \lambda \mathbf{I})\Delta \mathbf{x} = \mathbf{b}$$

$$\mathbf{x} \leftarrow \mathbf{x} + \Delta \mathbf{x} \quad \text{where, } \Delta \mathbf{x} = [\delta \xi^T, \delta a, \delta b]_{1 \times 8}^T$$

이 때 $\mathbf{x}_{1:6} \in \text{SE}(3)$ 는 카메라의 포즈를 의미하므로 증분량 $\Delta \mathbf{x}_{1:6}^\wedge \in \text{se}(3)$ 는 실제로 아래와 같이 업데이트 된다.

$$\mathbf{x}_{1:6} \leftarrow \exp(\Delta \mathbf{x}_{1:6}^\wedge) \mathbf{x}_{1:6}$$

$\mathbf{x}_{7:8}$ 는 Photometric parameter a,b를 의미하므로 아래와 같이 업데이트 된다.

$$\mathbf{x}_{7:8} \leftarrow \mathbf{x}_{7:8} + \Delta \mathbf{x}_{7:8}$$

2.1. Frames → Pose Tracking

\mathbf{H}, \mathbf{b} 를 각각 계산한 다음 Levenberg-Marquardt 방법을 통해 반복적으로 에러를 감소시킨다. 초기 $\lambda = 0.01$ 로 설정한다.

$$(\mathbf{H} + \lambda \mathbf{I})\Delta \mathbf{x} = \mathbf{b}$$

$$\mathbf{x} \leftarrow \mathbf{x} + \Delta \mathbf{x} \quad \text{where, } \Delta \mathbf{x} = [\delta \xi^T, \delta a, \delta b]_{1 \times 8}^T$$

이 때 $\mathbf{x}_{1:6} \in \text{SE}(3)$ 는 카메라의 포즈를 의미하므로 증분량 $\Delta \mathbf{x}_{1:6}^\wedge \in \text{se}(3)$ 는 실제로 아래와 같이 업데이트 된다.

$$\mathbf{x}_{1:6} \leftarrow \exp(\Delta \mathbf{x}_{1:6}^\wedge) \mathbf{x}_{1:6}$$

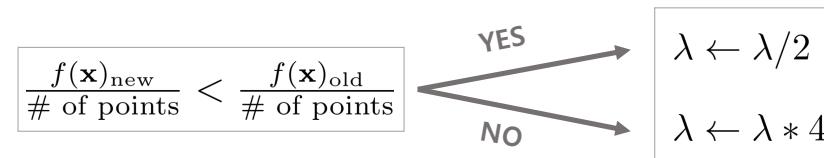
$\mathbf{x}_{7:8}$ 는 Photometric parameter a,b를 의미하므로 아래와 같이 업데이트 된다.

$$\mathbf{x}_{7:8} \leftarrow \mathbf{x}_{7:8} + \Delta \mathbf{x}_{7:8}$$

포즈가 업데이트되면 다시 한 번 에러를 계산한 후 에러가 충분히 감소하였는지 판단한다.

에러가 충분히 감소하였다면 나머지 후보 포즈들을 고려하지 않고 루프를 탈출한다. 그리고 lambda 값을 절반으로 줄인다.

만약 에러가 줄어들지 않았다면 lambda 값을 4배로 늘린다.



2.2. Frames → Keyframe Decision

매 프레임마다 Pose tracking이 끝나면 해당 프레임을 키프레임으로 결정할 지 판단한다.

2.2. Frames → Keyframe Decision

매 프레임마다 Pose tracking이 끝나면 해당 프레임을 키프레임으로 결정할 지 판단한다.

키프레임으로 결정하는 기준은 다음과 같다.

1. 해당 프레임이 가장 첫 번째 프레임인가?
2. (순수 회전을 제외한) 카메라 움직임으로 인한 Optical flow의 변화량 + Photometric (a,b)의 변화량이 일정 기준 이상인가?
3. Residual 값이 이전 키프레임과 비교했을 때 급격하게 변했는가?

2.2. Frames → Keyframe Decision

매 프레임마다 Pose tracking이 끝나면 해당 프레임을 키프레임으로 결정할 지 판단한다.

키프레임으로 결정하는 기준은 다음과 같다.

1. 해당 프레임이 가장 첫 번째 프레임인가?
2. (순수 회전을 제외한) 카메라 움직임으로 인한 Optical flow의 변화량 + Photometric (a,b)의 변화량이 일정 기준 이상인가?
3. Residual 값이 이전 키프레임과 비교했을 때 급격하게 변했는가?

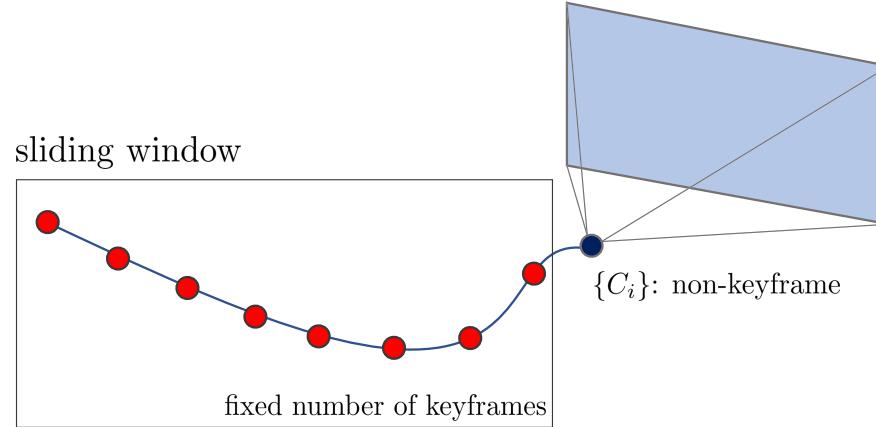
위 세 가지 기준 중 한가지라도 만족하게 되면 해당 프레임은 키프레임으로 간주된다.

3. Non-Keyframes

3.1. Non-Keyframes → Inverse Depth Update

keyframe
frame

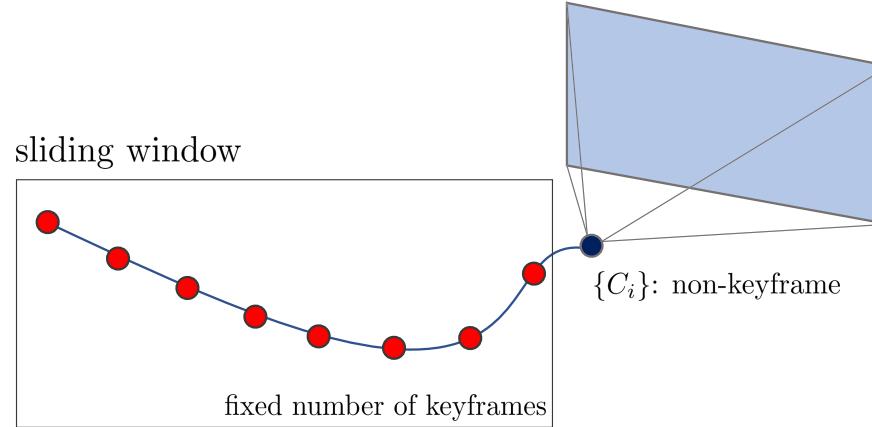
키프레임으로 간주되지 않은 경우, 해당 프레임은 현재 Sliding Window에 있는 키프레임들의 Immature point idepth 업데이트에 사용된다.



3.1. Non-Keyframes → Inverse Depth Update

● keyframe
● frame

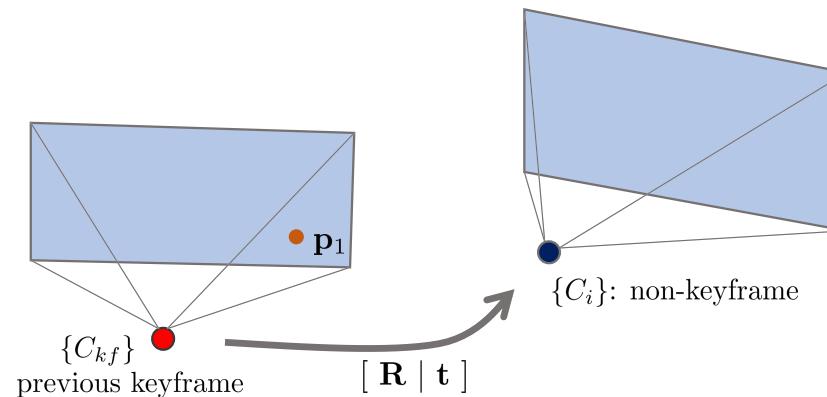
키프레임으로 간주되지 않은 경우, 해당 프레임은 현재 Sliding Window에 있는 키프레임들의 Immature point idepth 업데이트에 사용된다.



Immature point idepth update

Immature point idepth 업데이트의 자세한 과정은 다음과 같다.

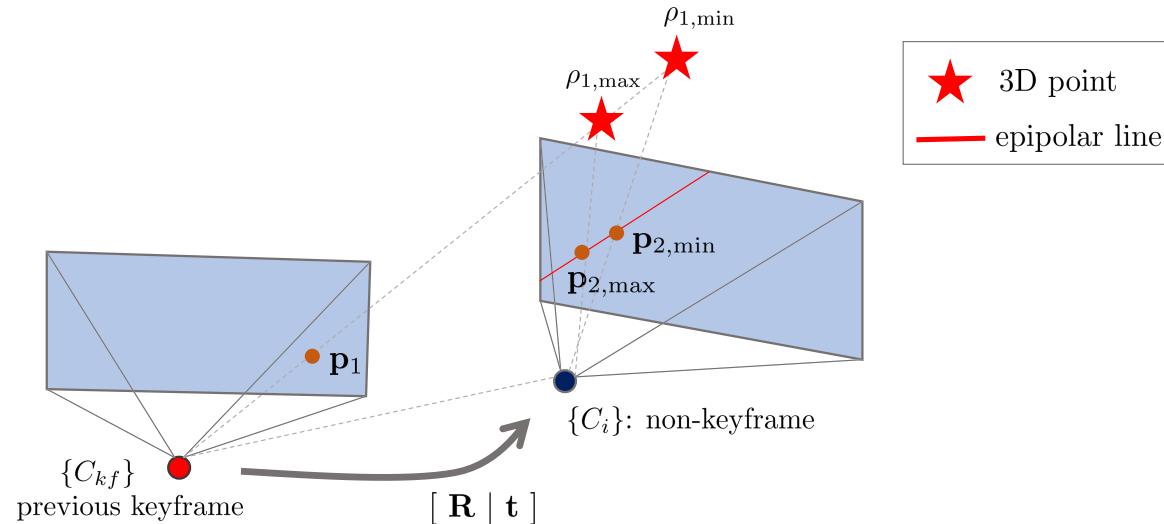
- 1) 우선, 이전 키프레임에 존재하는 idepth를 알고 있는 Immature point들을 상대 포즈 $[R | t]$ 을 기반으로 현재 프레임에 프로젝션한다.



3.1. Non-Keyframes → Inverse Depth Update

Immature point idepth update

2) 이 때, \mathbf{p}_1 을 한 번만 프로젝션 하지 않고 Epipolar line을 찾기 위해 idepth 값을 변경해서 총 2번 프로젝션한다.



$$\mathbf{p}_{2r} = \mathbf{KRK}^{-1}\mathbf{p}_1$$

$$\mathbf{p}_{2,\min} = \mathbf{p}_{2r} + \mathbf{Kt}\rho_{1,\min}$$

$$\mathbf{p}_{2,\max} = \mathbf{p}_{2r} + \mathbf{Kt}\rho_{1,\max}$$

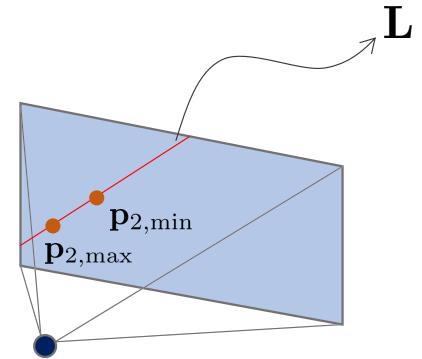
$\rho_{1,\max} 0$ | 기준에 설정되어 있지 않다면
Epipolar line을 찾기 위해 임의로 설정한 값을 사용한다. ($\rho_{1,\max} = 0.01$)

3.1. Non-Keyframes → Inverse Depth Update

Immature point idepth update

3) $\frac{\mathbf{p}_{2,\text{max}} - \mathbf{p}_{2,\text{min}}}{\|\mathbf{p}_{2,\text{max}} - \mathbf{p}_{2,\text{min}}\|}$ 값을 계산해서 Epipolar Line의 방향을 구한다. Epipolar Line을 수식으로 표현하면 다음과 같다.

$$\mathbf{L} := \{\mathbf{l}_0 + \lambda [l_x \ l_y]^T\}$$



3.1. Non-Keyframes → Inverse Depth Update

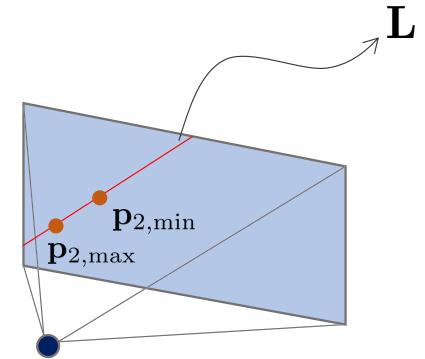
Immature point idepth update

3) $\frac{\mathbf{p}_{2,\max} - \mathbf{p}_{2,\min}}{\|\mathbf{p}_{2,\max} - \mathbf{p}_{2,\min}\|}$ 값을 계산해서 Epipolar Line의 방향을 구한다. Epipolar Line을 수식으로 표현하면 다음과 같다.

$$\mathbf{L} := \{\mathbf{l}_0 + \lambda[l_x \ l_y]^T\}$$

이 때, $\mathbf{l}_0 = \mathbf{p}_{2,\min}$ 는 원점을 의미하고 $[l_x \ l_y]^T = \frac{\mathbf{p}_{2,\max} - \mathbf{p}_{2,\min}}{\|\mathbf{p}_{2,\max} - \mathbf{p}_{2,\min}\|}$ 는 방향을 의미한다.

λ 는 discrete search step size를 의미한다.



3.1. Non-Keyframes → Inverse Depth Update

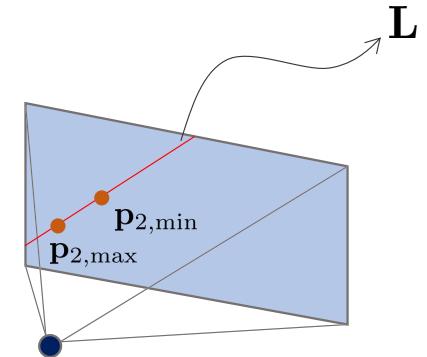
Immature point idepth update

3) $\frac{\mathbf{p}_{2,\max} - \mathbf{p}_{2,\min}}{\|\mathbf{p}_{2,\max} - \mathbf{p}_{2,\min}\|}$ 값을 계산해서 Epipolar Line의 방향을 구한다. Epipolar Line을 수식으로 표현하면 다음과 같다.

$$\mathbf{L} := \{\mathbf{l}_0 + \lambda[l_x \ l_y]^T\}$$

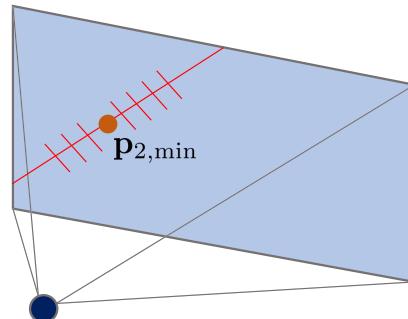
이 때, $\mathbf{l}_0 = \mathbf{p}_{2,\min}$ 는 원점을 의미하고 $[l_x \ l_y]^T = \frac{\mathbf{p}_{2,\max} - \mathbf{p}_{2,\min}}{\|\mathbf{p}_{2,\max} - \mathbf{p}_{2,\min}\|}$ 는 방향을 의미한다.

λ 는 discrete search step size를 의미한다.

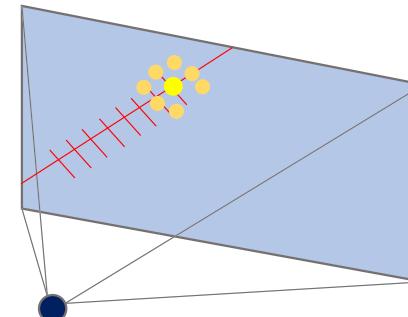


4) 다음으로 maximum discrete search 범위를 정한 후 \mathbf{L} 위에서 가장 밝기 오차가 작은 픽셀을 선정한다.

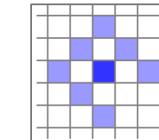
이 때도 패치 기반으로 밝기 오차를 계산한다.



Set the discrete search range



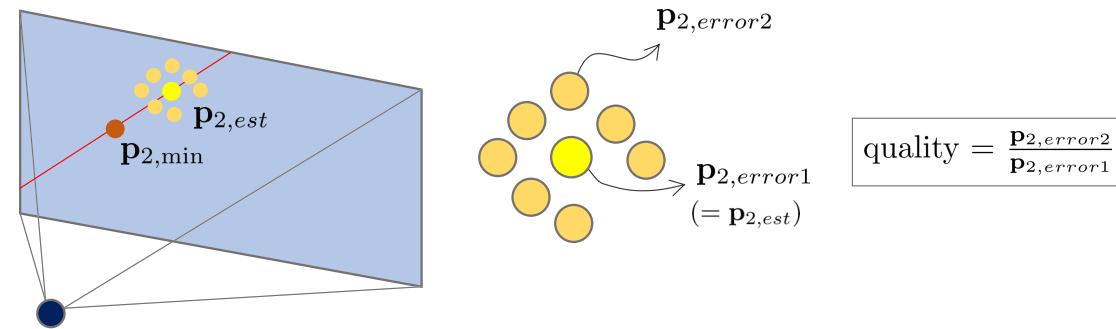
Find two smallest errors



3.1. Non-Keyframes → Inverse Depth Update

Immature point idepth update

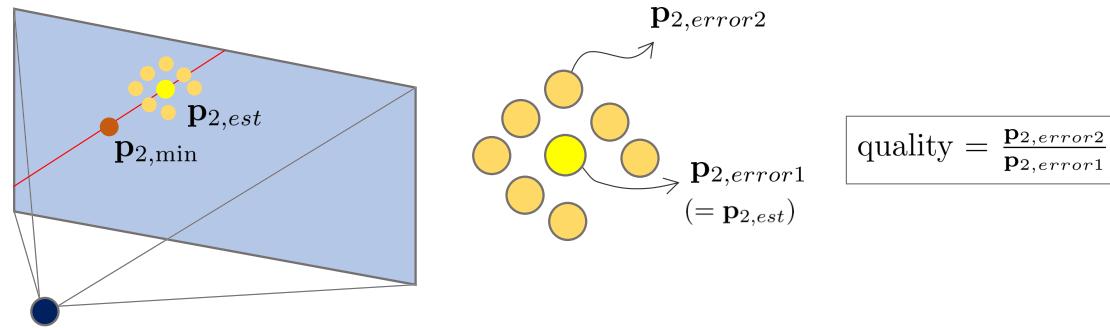
5) 밝기 오차가 가장 작은 점을 선정했으면 패치 내에서 밝기 오차가 두 번째로 작은 점을 선정한 후 두 오차의 비율을 통해 해당 점의 퀄리티를 계산한다.



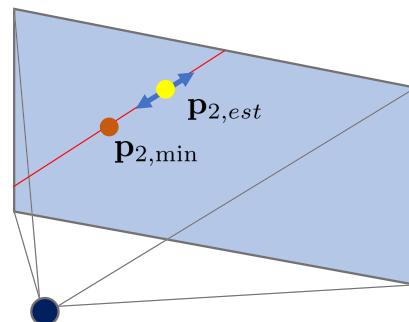
3.1. Non-Keyframes → Inverse Depth Update

Immature point idepth update

5) 밝기 오차가 가장 작은 점을 선정했으면 패치 내에서 밝기 오차가 두 번째로 작은 점을 선정한 후 두 오차의 비율을 통해 해당 점의 퀄리티를 계산한다.



6) 해당 $\mathbf{p}_{2,error1}$ 점은 discrete search를 통해 찾은 값이므로 보다 정교한 위치를 찾기 위해 Gauss-Newton optimization을 수행하여 최적의 위치 \mathbf{p}_2^* 를 구한다.



Perform Gauss-Newton optimization for refinement

3.1. Non-Keyframes → Inverse Depth Update

Immature point idepth update

7) 최적화 과정을 통해 최적의 위치 \mathbf{p}_2^* 를 찾았으면 다음으로 해당 픽셀의 idepth 값을 업데이트해준다.

3.1. Non-Keyframes → Inverse Depth Update

Immature point iddepth update

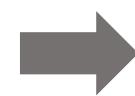
7) 최적화 과정을 통해 최적의 위치 \mathbf{p}_2^* 를 찾았으면 다음으로 해당 픽셀의 iddepth 값을 업데이트해준다.

iddepth 값을 업데이트하기 위해 해당 점을 수식으로 전개하면 다음과 같다.

$$\begin{aligned}\mathbf{p}_{2r} &= \mathbf{KRK}^{-1}\mathbf{p}_1 \\ \mathbf{p}_2 &= \mathbf{p}_{2r} + \mathbf{Kt}\rho_1\end{aligned}$$

를 전개하면

$$\begin{aligned}\mathbf{p}_{2r} &= \mathbf{KRK}^{-1}[u_1 \ v_1 \ 1]^T = [m_1 \ m_2 \ m_3]^T \\ \mathbf{Kt} &= [n_1 \ n_2 \ n_3]^T\end{aligned}$$



$$\mathbf{p}_2 = \begin{bmatrix} u_2 \\ v_2 \\ 1 \end{bmatrix} \quad u_2 = \frac{m_1+n_1\rho_1}{m_3+n_3\rho_3} \quad v_2 = \frac{m_2+n_2\rho_1}{m_3+n_3\rho_3}$$

이 된다.

3.1. Non-Keyframes → Inverse Depth Update

Immature point iddepth update

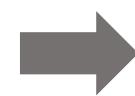
7) 최적화 과정을 통해 최적의 위치 \mathbf{p}_2^* 를 찾았으면 다음으로 해당 픽셀의 iddepth 값을 업데이트해준다.

iddepth 값을 업데이트하기 위해 해당 점을 수식으로 전개하면 다음과 같다.

$$\begin{aligned}\mathbf{p}_{2r} &= \mathbf{KRK}^{-1}\mathbf{p}_1 \\ \mathbf{p}_2 &= \mathbf{p}_{2r} + \mathbf{Kt}\rho_1\end{aligned}$$

를 전개하면

$$\begin{aligned}\mathbf{p}_{2r} &= \mathbf{KRK}^{-1}[u_1 \ v_1 \ 1]^T = [m_1 \ m_2 \ m_3]^T \\ \mathbf{Kt} &= [n_1 \ n_2 \ n_3]^T\end{aligned}$$



$$\mathbf{p}_2 = \begin{bmatrix} u_2 \\ v_2 \\ 1 \end{bmatrix} \quad \begin{aligned} u_2 &= \frac{m_1+n_1\rho_1}{m_3+n_3\rho_3} \\ v_2 &= \frac{m_2+n_2\rho_1}{m_3+n_3\rho_3} \end{aligned}$$

이 된다.

이를 iddepth를 기준으로 다시 정리하면 다음과 같은 2개의 식이 유도된다.

$$\begin{aligned}\rho_1 &= \frac{m_3u_2-m_1}{n_1-n_3u_2} \\ \rho_1 &= \frac{m_3v_2-m_2}{n_2-n_3v_2}\end{aligned}$$

3.1. Non-Keyframes → Inverse Depth Update

Immature point idepth update

8) 다음과 같이 유도된 식에서 최적의 위치 \mathbf{p}_2^* 를 넣고 다시 전개하면 다음과 같다.

x gradient가 큰 경우 $\Delta u > \Delta v$

$$\rho_{1,\min} = \frac{m_3(u_2^* - \alpha\Delta u) - m_1}{n_1 - n_3(u_2^* - \alpha\Delta u)}$$
$$\rho_{1,\max} = \frac{m_3(u_2^* + \alpha\Delta u) - m_1}{n_1 - n_3(u_2^* + \alpha\Delta u)}$$

y gradient가 큰 경우 $\Delta u < \Delta v$

$$\rho_{1,\min} = \frac{m_3(v_2^* - \alpha\Delta v) - m_2}{n_2 - n_3(v_2^* - \alpha\Delta v)}$$
$$\rho_{1,\max} = \frac{m_3(v_2^* + \alpha\Delta v) - m_2}{n_2 - n_3(v_2^* + \alpha\Delta v)}$$

3.1. Non-Keyframes → Inverse Depth Update

Immature point idepth update

8) 다음과 같이 유도된 식에서 최적의 위치 \mathbf{p}_2^* 를 넣고 다시 전개하면 다음과 같다.

x gradient가 큰 경우 $\Delta u > \Delta v$

$$\rho_{1,\min} = \frac{m_3(u_2^* - \alpha\Delta u) - m_1}{n_1 - n_3(u_2^* - \alpha\Delta u)}$$

$$\rho_{1,\max} = \frac{m_3(u_2^* + \alpha\Delta u) - m_1}{n_1 - n_3(u_2^* + \alpha\Delta u)}$$

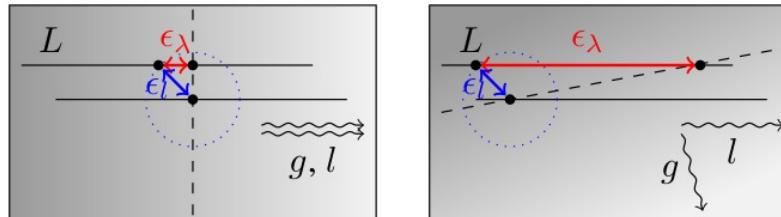
y gradient가 큰 경우 $\Delta u < \Delta v$

$$\rho_{1,\min} = \frac{m_3(v_2^* - \alpha\Delta v) - m_2}{n_2 - n_3(v_2^* - \alpha\Delta v)}$$

$$\rho_{1,\max} = \frac{m_3(v_2^* + \alpha\Delta v) - m_2}{n_2 - n_3(v_2^* + \alpha\Delta v)}$$

여기서 α 값은 gradient 방향과 Epipolar line의 방향이 수직일수록 1에서부터 값이 증가하는 가중치를 의미한다.

해당 α 값이 너무 커지면 두 방향이 거의 수직에 가깝다고 판단하여 idepth 업데이트를 중단한다.



<https://jsturm.de/publications/data/engel2013iccv.pdf> 논문 참조

3.1. Non-Keyframes → Inverse Depth Update

Immature point idepth update

8) 다음과 같이 유도된 식에서 최적의 위치 \mathbf{p}_2^* 를 넣고 다시 전개하면 다음과 같다.

x gradient가 큰 경우 $\Delta u > \Delta v$

$$\rho_{1,\min} = \frac{m_3(u_2^* - \alpha\Delta u) - m_1}{n_1 - n_3(u_2^* - \alpha\Delta u)}$$

$$\rho_{1,\max} = \frac{m_3(u_2^* + \alpha\Delta u) - m_1}{n_1 - n_3(u_2^* + \alpha\Delta u)}$$

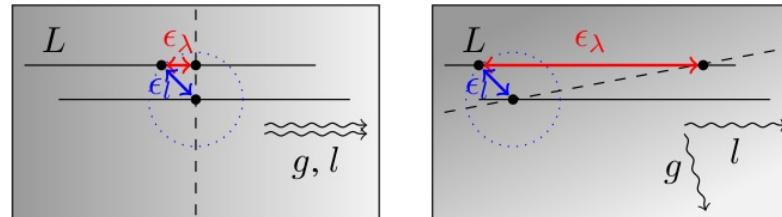
y gradient가 큰 경우 $\Delta u < \Delta v$

$$\rho_{1,\min} = \frac{m_3(v_2^* - \alpha\Delta v) - m_2}{n_2 - n_3(v_2^* - \alpha\Delta v)}$$

$$\rho_{1,\max} = \frac{m_3(v_2^* + \alpha\Delta v) - m_2}{n_2 - n_3(v_2^* + \alpha\Delta v)}$$

여기서 α 값은 gradient 방향과 Epipolar line의 방향이 수직일수록 1에서부터 값이 증가하는 가중치를 의미한다.

해당 α 값이 너무 커지면 두 방향이 거의 수직에 가깝다고 판단하여 idepth 업데이트를 중단한다.



<https://jsturm.de/publications/data/engel2013iccv.pdf> 논문 참조

이와 같은 1)~8) 과정을 통해 이전 키프레임에서 Immature point \mathbf{p}_1 의 idepth 값을 업데이트할 수 있다.

4. Keyframes

4.1. Keyframes → Inverse Depth Update

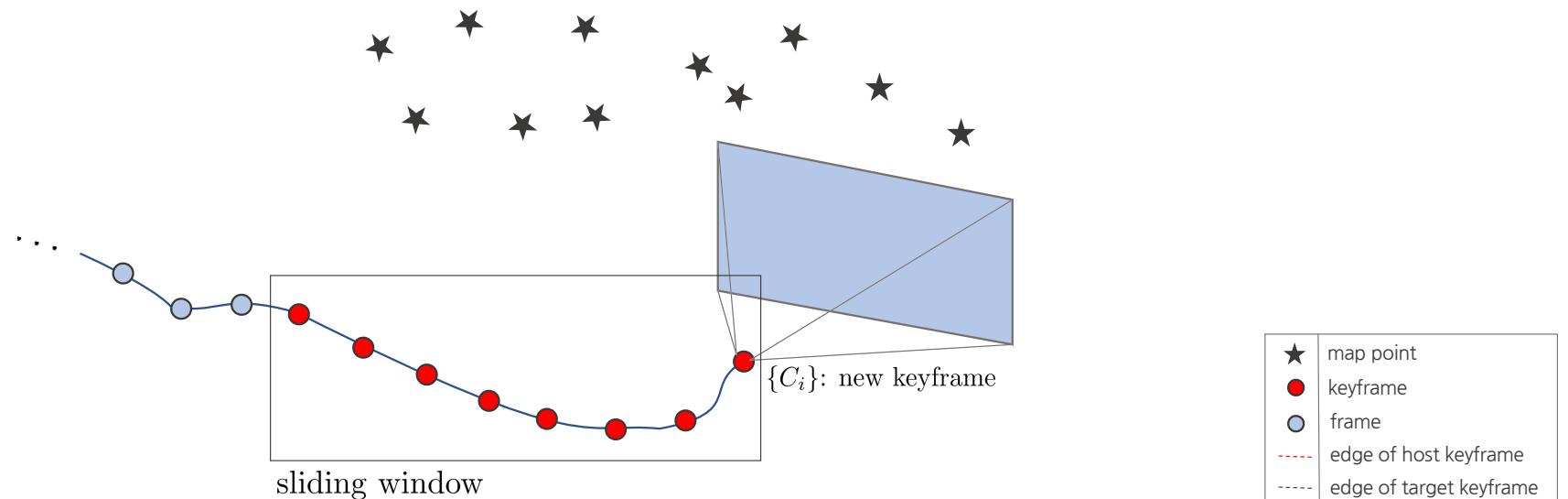
To Be Added

4.2. Keyframes → Immature Point Activation

To Be Added

4.3.1. Keyframes → Sliding Window Optimization → Error Function Formulation

특정 프레임이 키프레임으로 결정되면 Sliding Window 내부에 키프레임들과 새로운 키프레임 사이에 에러함수를 업데이트해야 한다.
이 때, 키프레임 뿐만 아니라 이와 연결되어 있는 맵 상의 포인트들까지 같이 최적화하는 **Local Bundle Adjustment**를 수행한다.



4.3.1. Keyframes → Sliding Window Optimization → Error Function Formulation

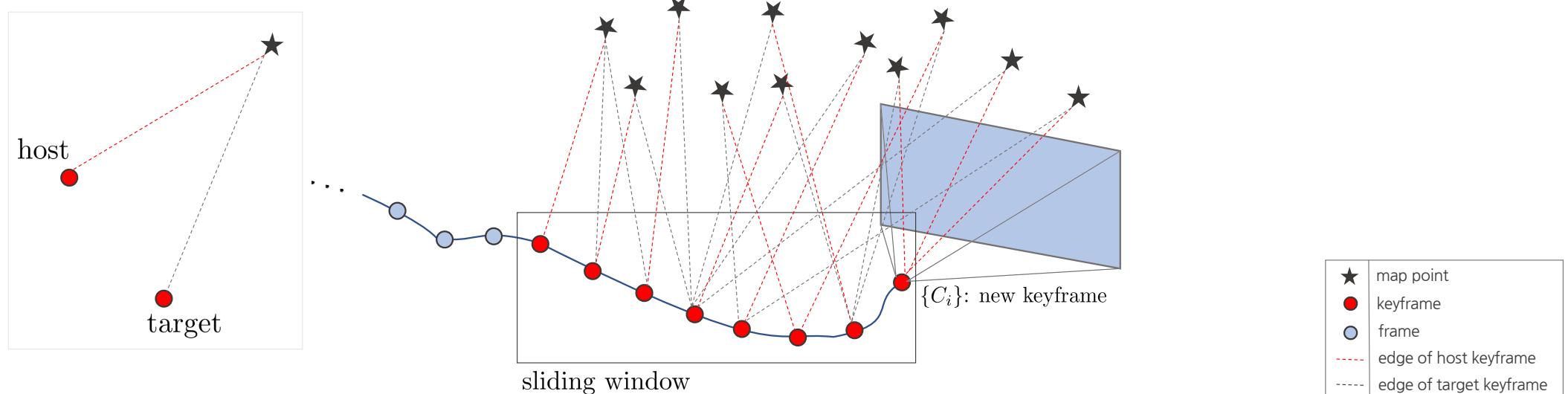
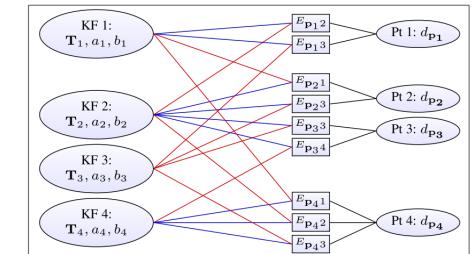
특정 프레임이 키프레임으로 결정되면 Sliding Window 내부에 키프레임들과 새로운 키프레임 사이에 에러함수를 업데이트해야 한다.

이 때, 키프레임 뿐만 아니라 이와 연결되어 있는 맵 상의 포인트들까지 같이 최적화하는 **Local Bundle Adjustment**를 수행한다.

주목할 점은 하나의 맵포인트가 두 개의 키프레임(host, target)과 연결되어 최적화된다는 점이다.

여기서 **host** 키프레임은 해당 맵포인트를 가장 먼저 발견한 키프레임이고 **target** 키프레임은 이후에 발견한 키프레임을 의미한다.

DSO paper Figure 5
blue: host, red: target



4.3.1. Keyframes → Sliding Window Optimization → Error Function Formulation

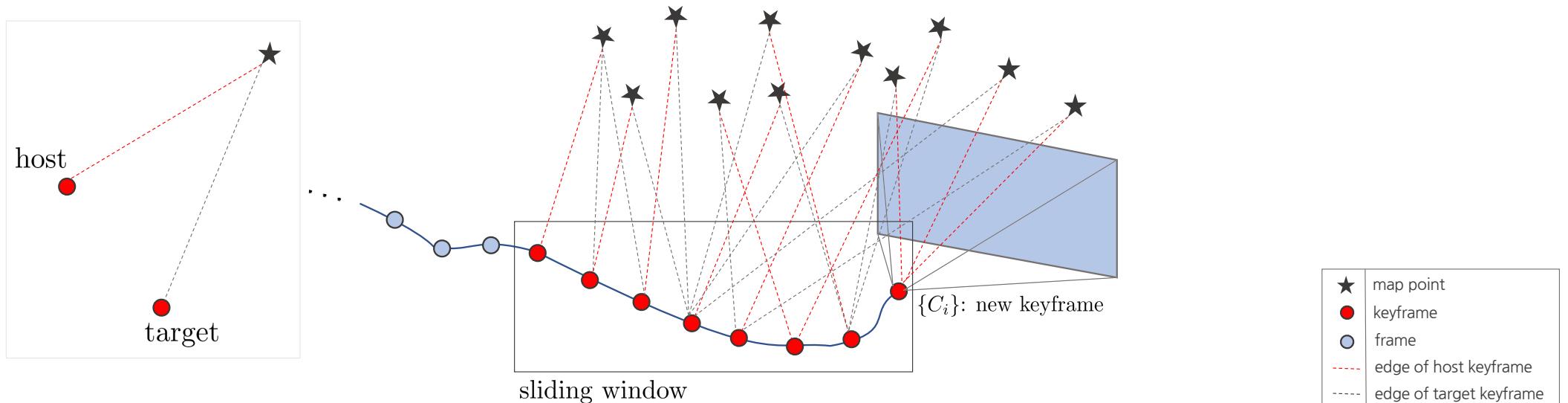
특정 프레임이 키프레임으로 결정되면 Sliding Window 내부에 키프레임들과 새로운 키프레임 사이에 에러함수를 업데이트해야 한다. 이 때, 키프레임 뿐만 아니라 이와 연결되어 있는 맵 상의 포인트들까지 같이 최적화하는 **Local Bundle Adjustment**를 수행한다.

주목할 점은 하나의 맵포인트가 두 개의 키프레임(host, target)과 연결되어 최적화된다는 점이다.

여기서 **host** 키프레임은 해당 맵포인트를 가장 먼저 발견한 키프레임이고 **target** 키프레임은 이후에 발견한 키프레임을 의미한다.

또한 최적화 과정에서 연산량을 줄이기 위해 Jacobian을 계산할 때 카메라의 월드좌표계를 기준으로 계산한 Jacobian $\frac{\partial \mathbf{r}}{\partial \xi_{wh}}$, $\frac{\partial \mathbf{r}}{\partial \xi_{wt}}$ 를 사용하지 않고 host, target 키프레임 간 상대 포즈를 기준으로 계산한 Jacobian $\frac{\partial \mathbf{r}}{\partial \xi_{th}}$ 을 사용한다.

따라서 해당 Jacobian을 사용하여 두 키프레임의 월드좌표계 기반 카메라 포즈 \mathbf{T}_{wh} , \mathbf{T}_{wt} 를 최적화 하기 위해 **Adjoint Transformation**을 사용한다.



4.3.1. Keyframes → Sliding Window Optimization → Error Function Formulation

Sliding Window Optimization을 수행하기 위해 우선 에러 함수를 설정해야 한다.

4.3.1. Keyframes → Sliding Window Optimization → Error Function Formulation

Sliding Window Optimization을 수행하기 위해 우선 에러 함수를 설정해야 한다.

이 때 에러 함수의 파라미터는 Pose(6) + Photometric(2) + Camera Intrinsics(4) + ldepth(N)로써 총 $12+N$ 개의 파라미터가 최적화에 사용된다.

참고로 Initialization 과정에서는 Pose(6) + Photometric(2) + ldepth(N)의 $8+N$ 개의 파라미터를 최적화하였다.

즉, Initialization 과정에 추가적으로 Camera Intrinsics (f_x, f_y, c_x, c_y)를 같이 최적화 했으므로 에러 함수 또한 Initialization 과정과 동일하다.

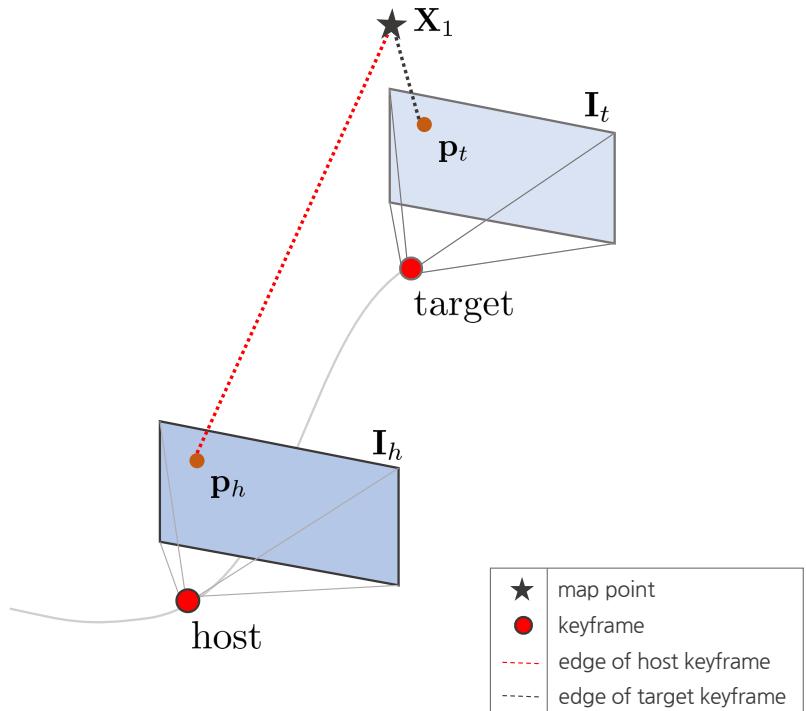
4.3.1. Keyframes → Sliding Window Optimization → Error Function Formulation

Sliding Window Optimization을 수행하기 위해 우선 에러 함수를 설정해야 한다.

이 때 에러 함수의 파라미터는 Pose(6) + Photometric(2) + Camera Intrinsics(4) + ldepth(N)로써 총 $12+N$ 개의 파라미터가 최적화에 사용된다.

참고로 Initialization 과정에서는 Pose(6) + Photometric(2) + ldepth(N)의 $8+N$ 개의 파라미터를 최적화하였다.

즉, Initialization 과정에 추가적으로 Camera Intrinsics (f_x, f_y, c_x, c_y)를 같이 최적화 했으므로 에러 함수 또한 Initialization 과정과 동일하다.



residual(=에러)는 다음과 같이 설정할 수 있다.

Initialization 단계와 다르게 초기값 b_0 추가됨

$$\mathbf{r} = \mathbf{I}_t(\mathbf{p}_t) - \exp(a)(\mathbf{I}_h(\mathbf{p}_h) - b_0) - b$$

where, $\mathbf{p}_t = \pi(\exp(\xi_{th}^\wedge)\mathbf{X}_1) = \pi(\exp(\xi_{th}^\wedge)\pi^{-1}(\mathbf{p}_h))$

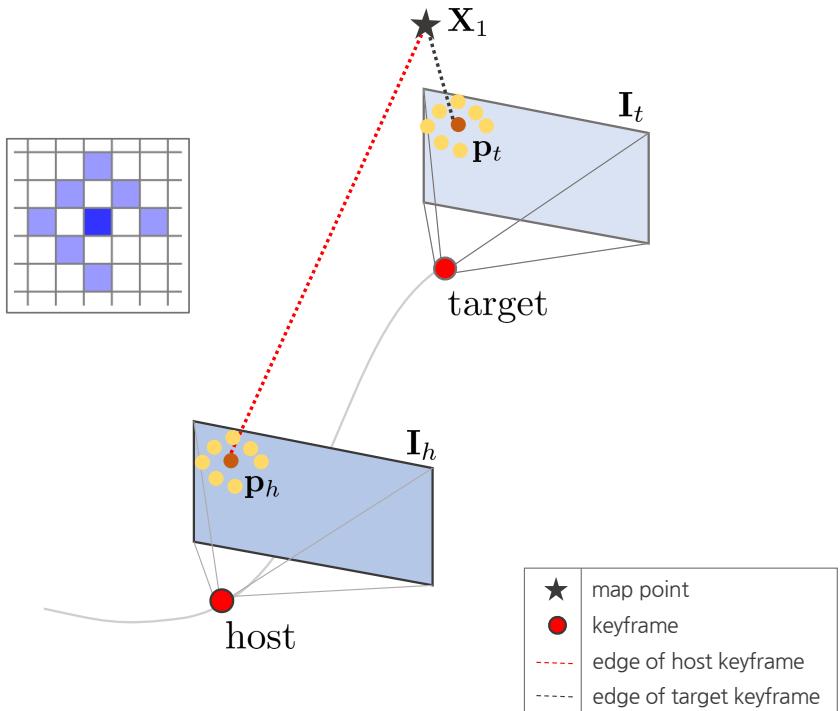
4.3.1. Keyframes → Sliding Window Optimization → Error Function Formulation

Sliding Window Optimization을 수행하기 위해 우선 에러 함수를 설정해야 한다.

이 때 에러 함수의 파라미터는 Pose(6) + Photometric(2) + Camera Intrinsics(4) + ldepth(N)로써 총 $12+N$ 개의 파라미터가 최적화에 사용된다.

참고로 Initialization 과정에서는 Pose(6) + Photometric(2) + ldepth(N)의 $8+N$ 개의 파라미터를 최적화하였다.

즉, Initialization 과정에 추가적으로 Camera Intrinsics (f_x, f_y, c_x, c_y)를 같이 최적화 했으므로 에러 함수 또한 Initialization 과정과 동일하다.



residual(=에러)는 다음과 같이 설정할 수 있다.

$$\mathbf{r} = \mathbf{I}_t(\mathbf{p}_t) - \exp(a)(\mathbf{I}_h(\mathbf{p}_h) - b_0) - b$$

where, $\mathbf{p}_t = \pi(\exp(\xi_{th}^\wedge)\mathbf{X}_1) = \pi(\exp(\xi_{th}^\wedge)\pi^{-1}(\mathbf{p}_h))$

이 때, residual은 한 점이 아닌 주변 점들을 포함한 8개 점들 에러의 합을 의미한다.

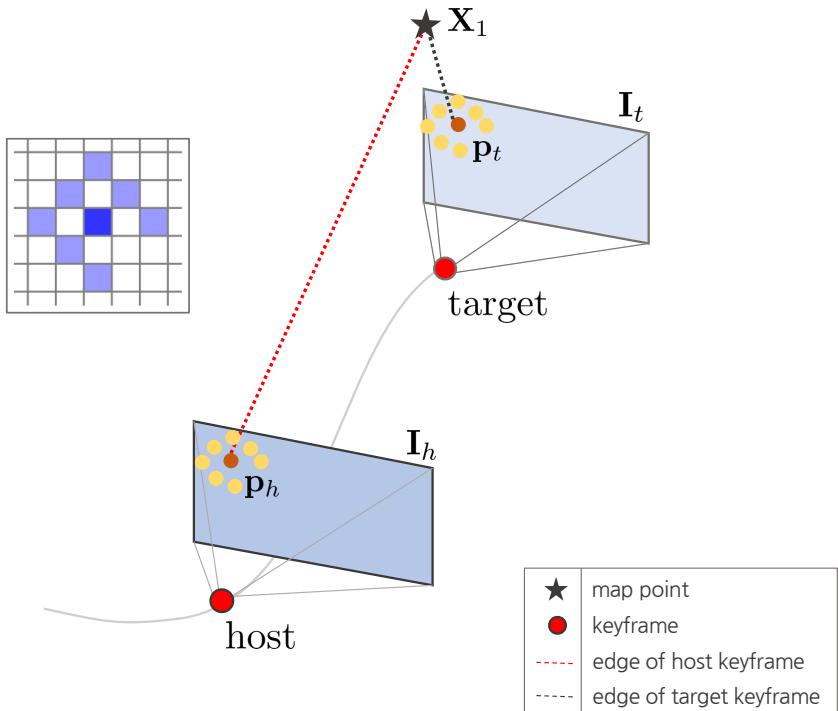
4.3.1. Keyframes → Sliding Window Optimization → Error Function Formulation

Sliding Window Optimization을 수행하기 위해 우선 에러 함수를 설정해야 한다.

이 때 에러 함수의 파라미터는 Pose(6) + Photometric(2) + Camera Intrinsics(4) + ldepth(N)로써 총 $12+N$ 개의 파라미터가 최적화에 사용된다.

참고로 Initialization 과정에서는 Pose(6) + Photometric(2) + ldepth(N)의 $8+N$ 개의 파라미터를 최적화하였다.

즉, Initialization 과정에 추가적으로 Camera Intrinsics (f_x, f_y, c_x, c_y)를 같이 최적화 했으므로 에러 함수 또한 Initialization 과정과 동일하다.



residual(=에러)는 다음과 같이 설정할 수 있다.

$$\mathbf{r} = \mathbf{I}_t(\mathbf{p}_t) - \exp(a)(\mathbf{I}_h(\mathbf{p}_h) - b_0) - b$$

where, $\mathbf{p}_t = \pi(\exp(\xi_{th}^\wedge)\mathbf{X}_1) = \pi(\exp(\xi_{th}^\wedge)\pi^{-1}(\mathbf{p}_h))$

이 때, residual은 한 점이 아닌 주변 점들을 포함한 8개 점들 에러의 합을 의미한다.

최종적으로 huber norm까지 적용된 에러함수는 다음과 같다.

$$H(\mathbf{r}) = w_h \mathbf{r}^2$$

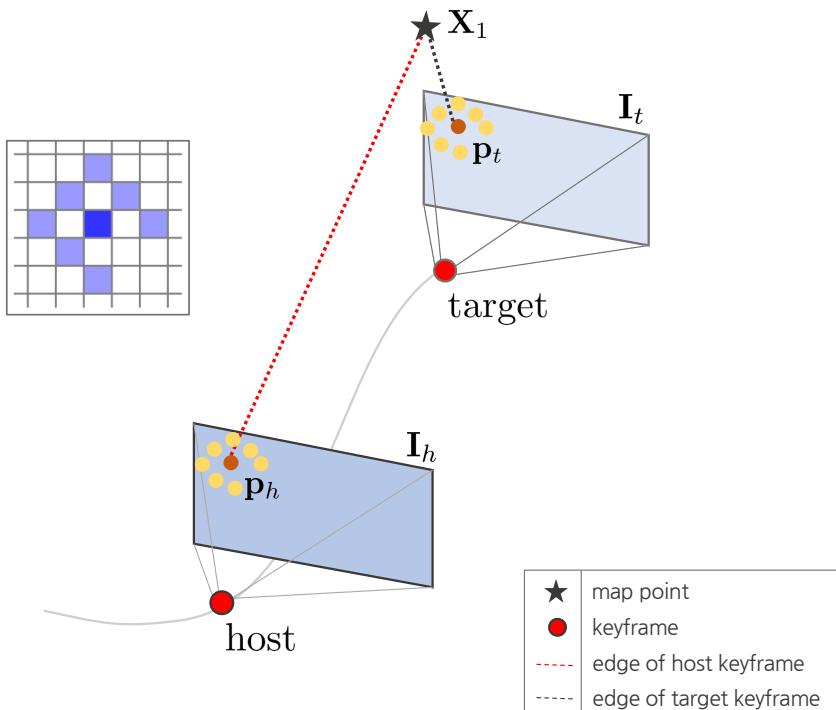
4.3.1. Keyframes → Sliding Window Optimization → Error Function Formulation

Sliding Window Optimization을 수행하기 위해 우선 에러 함수를 설정해야 한다.

이 때 에러 함수의 파라미터는 Pose(6) + Photometric(2) + Camera Intrinsics(4) + ldepth(N)로써 총 $12+N$ 개의 파라미터가 최적화에 사용된다.

참고로 Initialization 과정에서는 Pose(6) + Photometric(2) + ldepth(N)의 $8+N$ 개의 파라미터를 최적화하였다.

즉, Initialization 과정에 추가적으로 Camera Intrinsics (f_x, f_y, c_x, c_y)를 같이 최적화 했으므로 에러 함수 또한 Initialization 과정과 동일하다.



residual(=에러)는 다음과 같이 설정할 수 있다.

$$\mathbf{r} = \mathbf{I}_t(\mathbf{p}_t) - \exp(a)(\mathbf{I}_h(\mathbf{p}_h) - b_0) - b$$

where, $\mathbf{p}_t = \pi(\exp(\xi_{th}^\wedge)\mathbf{X}_1) = \pi(\exp(\xi_{th}^\wedge)\pi^{-1}(\mathbf{p}_h))$

이 때, residual은 한 점이 아닌 주변 점들을 포함한 8개 점들 에러의 합을 의미한다.

최종적으로 huber norm까지 적용된 에러함수는 다음과 같다.

$$H(\mathbf{r}) = w_h \mathbf{r}^2$$

해당 에러함수가 최소가 되는 파라미터를 찾기 위해 $f(\mathbf{x})^T f(\mathbf{x}) = H(\mathbf{r})$ 를 만족하는 $f(\mathbf{x})$ 를 최소화하는 문제로 변경하면 다음과 같다.

$$f(\mathbf{x}) = \sqrt{w_h} \mathbf{r} = \sqrt{w_h}(\mathbf{I}_2(\mathbf{p}_2) - \exp(a)(\mathbf{I}_1(\mathbf{p}_1 - b_0)) - b)$$

4.3.1. Keyframes → Sliding Window Optimization → Error Function Formulation

에러함수 $f(\mathbf{x})$ 의 state variable \mathbf{x} 는 다음과 같다.

$$f(\mathbf{x}) = \sqrt{w_h} \quad \mathbf{r} = \sqrt{w_h} (\mathbf{I}_t(\mathbf{p}_t) - \exp(a)(\mathbf{I}_h(\mathbf{p}_h - b_0)) - b)$$

$\mathbf{x} = [\rho_1^{(1)}, \dots, \rho_1^{(N)}, \delta\xi^T, a, b, \mathbf{c}]_{(N+12) \times 1}^T$	$\rho_1^{(i)}$ $\delta\xi$ a, b \mathbf{c}	1번 이미지에서 i번째 inverse depth 값 (N개) 두 카메라 간 상대 포즈 변화량 (twist) (6차원 벡터) (6개) 두 이미지 간 exposure time을 고려하여 이미지 밝기를 조절하는 파라미터 (2개) 카메라 내부 파라미터 (fx, fy, cx, cy) (4개)
---------------------------------------------------------------------------------------------------------	---------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------

4.3.1. Keyframes → Sliding Window Optimization → Error Function Formulation

에러함수 $f(\mathbf{x})$ 의 state variable \mathbf{x} 는 다음과 같다.

$$f(\mathbf{x}) = \sqrt{w_h} \quad \mathbf{r} = \sqrt{w_h} (\mathbf{I}_t(\mathbf{p}_t) - \exp(a)(\mathbf{I}_h(\mathbf{p}_h - b_0)) - b)$$

$$\mathbf{x} = [\rho_1^{(1)}, \dots, \rho_1^{(N)}, \delta\xi^T, a, b, \mathbf{c}]_{(N+12) \times 1}^T$$

$\rho_1^{(i)}$	1번 이미지에서 i번째 inverse depth 값 (N개)
$\delta\xi$	두 카메라 간 상대 포즈 변화량 (twist) (6차원 벡터) (6개)
a, b	두 이미지 간 exposure time을 고려하여 이미지 밝기를 조절하는 파라미터 (2개)
\mathbf{c}	카메라 내부 파라미터 (fx, fy, cx, cy) (4개)

Initialization 과정과 달리 state variable \mathbf{c} 가 포함된 것을 알 수 있다.

4.3.1. Keyframes → Sliding Window Optimization → Error Function Formulation

에러함수 $f(\mathbf{x})$ 의 state variable \mathbf{x} 는 다음과 같다.

$$f(\mathbf{x}) = \sqrt{w_h} \quad \mathbf{r} = \sqrt{w_h} (\mathbf{I}_t(\mathbf{p}_t) - \exp(a)(\mathbf{I}_h(\mathbf{p}_h - \mathbf{b}_0)) - \mathbf{b})$$

$\mathbf{x} = [\rho_1^{(1)}, \dots, \rho_1^{(N)}, \delta\xi^T, a, b, \mathbf{c}]_{(N+12) \times 1}^T$	$\rho_1^{(i)}$	1번 이미지에서 i번째 inverse depth 값 (N개)
	$\delta\xi$	두 카메라 간 상대 포즈 변화량 (twist) (6차원 벡터) (6개)
	a, b	두 이미지 간 exposure time을 고려하여 이미지 밝기를 조절하는 파라미터 (2개)
	\mathbf{c}	카메라 내부 파라미터 (f_x, f_y, c_x, c_y) (4개)

Initialization 과정과 달리 state variable에 camera intrinsics \mathbf{c} 가 포함된 것을 알 수 있다.

Initialization 과정에서 이미 아래와 같이 최적화에 필요한 Jacobian들을 유도하였다.

하지만 Sliding Window Optimization 과정에서는 추가적으로 Jacobian of Camera Intrinsics $\frac{\partial f(\mathbf{x})}{\partial \mathbf{c}}$ 을 유도해야 한다.

idepth(N)

$$\frac{\partial f(\mathbf{x})}{\partial \rho_1} = \sqrt{w_h} \rho_1^{-1} \rho_2 (\nabla \mathbf{I}_x f_x (t_x - u'_2 t_z) + \nabla \mathbf{I}_y f_y (t_y - v'_2 t_z))$$

pose(6)

$$\frac{\partial f(\mathbf{x})}{\partial \delta\xi} = \sqrt{w_h} \begin{bmatrix} \nabla \mathbf{I}_x \rho_2 f_x \\ \nabla \mathbf{I}_y \rho_2 f_y \\ -\rho_2 (\nabla \mathbf{I}_x f_x u'_2 + \nabla \mathbf{I}_y f_y v'_2) \\ -\nabla \mathbf{I}_x f_x u'_2 v'_2 - \nabla \mathbf{I}_y f_y (1 + v'^2_2) \\ \nabla \mathbf{I}_x f_x (1 + u'^2_2) + \nabla \mathbf{I}_y f_y u'_2 v'_2 \\ -\nabla \mathbf{I}_x f_x v'_2 + \nabla \mathbf{I}_y f_y u'_2 \end{bmatrix}^T$$

photometric(2)

$$\begin{aligned} \frac{\partial f(\mathbf{x})}{\partial a} &= \sqrt{w_h} \exp(a) \mathbf{I}_1 (b_0 - \mathbf{p}_1) \\ \frac{\partial f(\mathbf{x})}{\partial b} &= -\sqrt{w_h} \end{aligned}$$

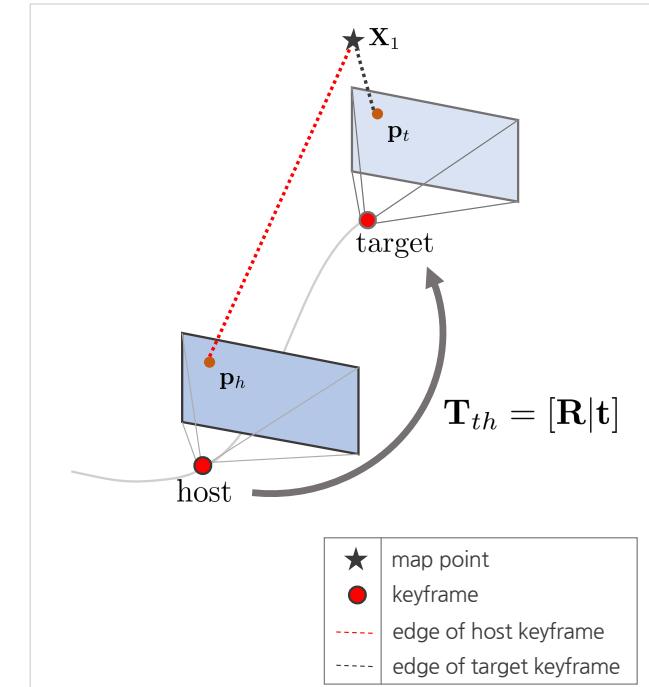
$$\begin{aligned} \frac{\partial \mathbf{r}}{\partial a} &= -\exp(a) \mathbf{I}_1(\mathbf{p}_1) \\ \frac{\partial \mathbf{r}}{\partial a} &= \exp(a) \mathbf{I}_1(b_0 - \mathbf{p}_1) \end{aligned}$$

Initialization 단계와 다르게 초기값 b0
추가됨

4.3.2. Keyframes → Sliding Window Optimization → Jacobian Derivation of Camera Intrinsics

Jacobian of Camera Intrinsics $\frac{\partial f(\mathbf{x})}{\partial \mathbf{c}}$ 는 다음과 같이 분해할 수 있다.

$$\frac{\partial f(\mathbf{x})}{\partial \mathbf{c}} = \frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_t} \frac{\partial \mathbf{p}_t}{\partial \mathbf{c}}$$



More detail: <https://blog.csdn.net/xxxinttp/article/details/90640350>

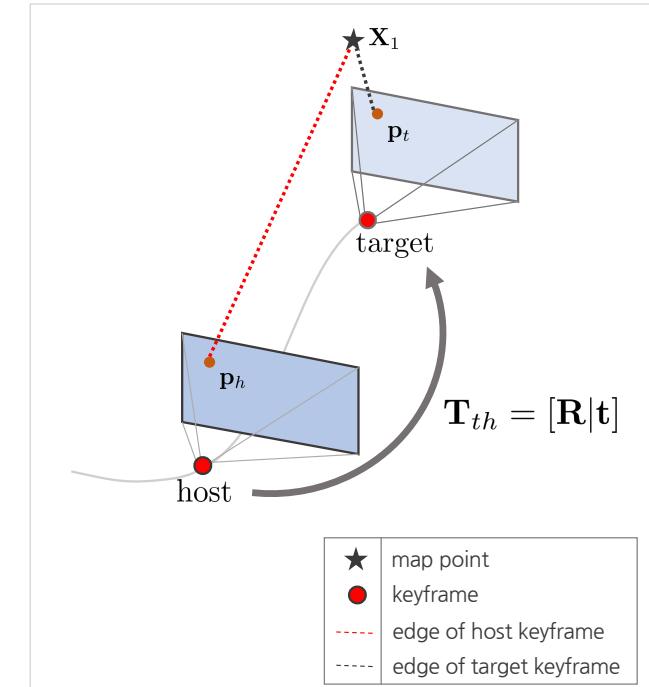
4.3.2. Keyframes → Sliding Window Optimization → Jacobian Derivation of Camera Intrinsics

Jacobian of Camera Intrinsics $\frac{\partial f(\mathbf{x})}{\partial \mathbf{c}}$ 는 다음과 같이 분해할 수 있다.

$$\frac{\partial f(\mathbf{x})}{\partial \mathbf{c}} = \frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_t} \frac{\partial \mathbf{p}_t}{\partial \mathbf{c}}$$

이 때, $f(\mathbf{x}) = \sqrt{w_h} \mathbf{r}$ 이므로 간결하게 residual 항으로 표현하면 다음과 같다.

$$\frac{\partial \mathbf{r}}{\partial \mathbf{c}} = \frac{\partial \mathbf{r}}{\partial \mathbf{p}_t} \frac{\partial \mathbf{p}_t}{\partial \mathbf{c}}$$



More detail: <https://blog.csdn.net/xxxinttp/article/details/90640350>

4.3.2. Keyframes → Sliding Window Optimization → Jacobian Derivation of Camera Intrinsics

Jacobian of Camera Intrinsics $\frac{\partial f(\mathbf{x})}{\partial \mathbf{c}}$ 는 다음과 같이 분해할 수 있다.

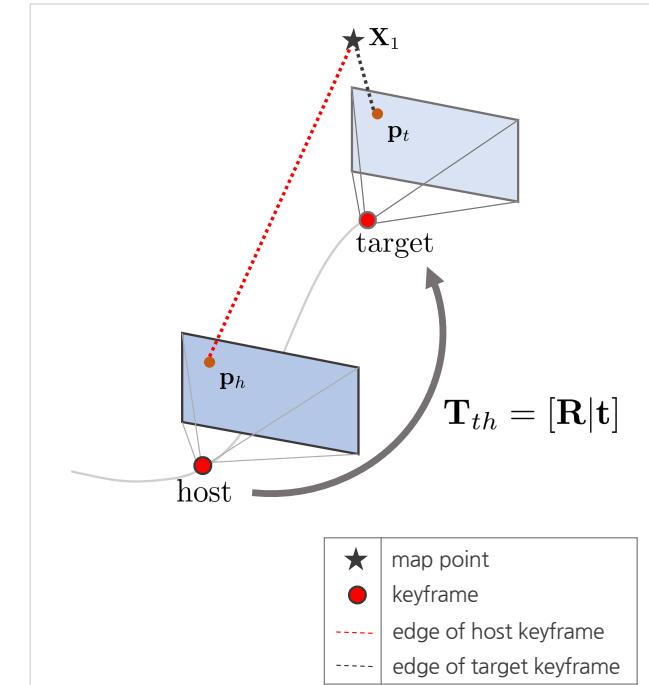
$$\frac{\partial f(\mathbf{x})}{\partial \mathbf{c}} = \frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_t} \frac{\partial \mathbf{p}_t}{\partial \mathbf{c}}$$

이 때, $f(\mathbf{x}) = \sqrt{w_h}$ \mathbf{r} 이므로 간결하게 residual 항으로 표현하면 다음과 같다.

$$\frac{\partial \mathbf{r}}{\partial \mathbf{c}} = \frac{\partial \mathbf{r}}{\partial \mathbf{p}_t} \frac{\partial \mathbf{p}_t}{\partial \mathbf{c}}$$

$\frac{\partial \mathbf{r}}{\partial \mathbf{p}_t}$ 는 Initialization 섹션에서 다음과 같이 이미 유도하였다.

$$\frac{\partial \mathbf{r}}{\partial \mathbf{p}_t} = \frac{\partial \mathbf{I}_t(\mathbf{p}_t)}{\partial \mathbf{p}_t} = [\nabla \mathbf{I}_x \quad \nabla \mathbf{I}_y]$$



4.3.2. Keyframes → Sliding Window Optimization → Jacobian Derivation of Camera Intrinsics

Jacobian of Camera Intrinsics $\frac{\partial f(\mathbf{x})}{\partial \mathbf{c}}$ 는 다음과 같이 분해할 수 있다.

$$\frac{\partial f(\mathbf{x})}{\partial \mathbf{c}} = \frac{\partial f(\mathbf{x})}{\partial \mathbf{p}_t} \frac{\partial \mathbf{p}_t}{\partial \mathbf{c}}$$

이 때, $f(\mathbf{x}) = \sqrt{w_h}$ \mathbf{r} 이므로 간결하게 residual 항으로 표현하면 다음과 같다.

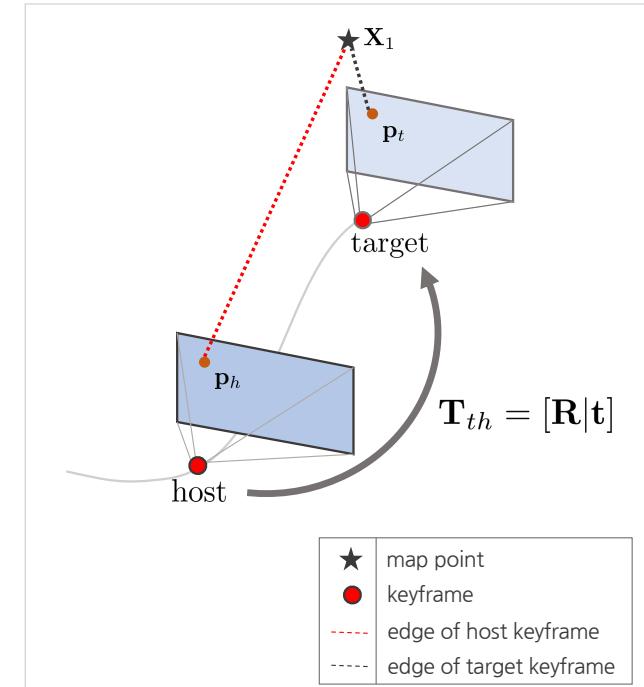
$$\frac{\partial \mathbf{r}}{\partial \mathbf{c}} = \frac{\partial \mathbf{r}}{\partial \mathbf{p}_t} \frac{\partial \mathbf{p}_t}{\partial \mathbf{c}}$$

$\frac{\partial \mathbf{r}}{\partial \mathbf{p}_t}$ 는 Initialization 섹션에서 다음과 같이 이미 유도하였다.

$$\frac{\partial \mathbf{r}}{\partial \mathbf{p}_t} = \frac{\partial \mathbf{I}_t(\mathbf{p}_t)}{\partial \mathbf{p}_t} = [\nabla \mathbf{I}_x \quad \nabla \mathbf{I}_y]$$

그리고 $\frac{\partial \mathbf{p}_t}{\partial \mathbf{c}}$ 는 다음과 같이 유도할 수 있다.

$$\begin{aligned} \frac{\partial \mathbf{p}_t}{\partial \mathbf{c}} &= \begin{bmatrix} \frac{\partial u_t}{\partial f_x} & \frac{\partial u_t}{\partial f_y} & \frac{\partial u_t}{\partial c_x} & \frac{\partial u_t}{\partial c_y} \\ \frac{\partial v_t}{\partial f_x} & \frac{\partial v_t}{\partial f_y} & \frac{\partial v_t}{\partial c_x} & \frac{\partial v_t}{\partial c_y} \end{bmatrix} \\ &= \begin{bmatrix} \bar{u}_t + f_x \frac{\partial \bar{u}_t}{\partial f_x} & f_x \frac{\partial \bar{u}_t}{\partial f_y} & f_x \frac{\partial \bar{u}_t}{\partial c_x} + 1 & f_x \frac{\partial \bar{u}_t}{\partial c_y} \\ f_y \frac{\partial \bar{v}_t}{\partial f_x} & \bar{v}_t + f_y \frac{\partial \bar{v}_t}{\partial f_y} & f_y \frac{\partial \bar{v}_t}{\partial c_x} & f_y \frac{\partial \bar{v}_t}{\partial c_y} + 1 \end{bmatrix} \quad \because u_t = f_x \bar{u}_t + c_x \\ &\qquad\qquad\qquad v_t = f_y \bar{v}_t + c_y \end{aligned}$$



4.3.2. Keyframes → Sliding Window Optimization → Jacobian Derivation of Camera Intrinsics

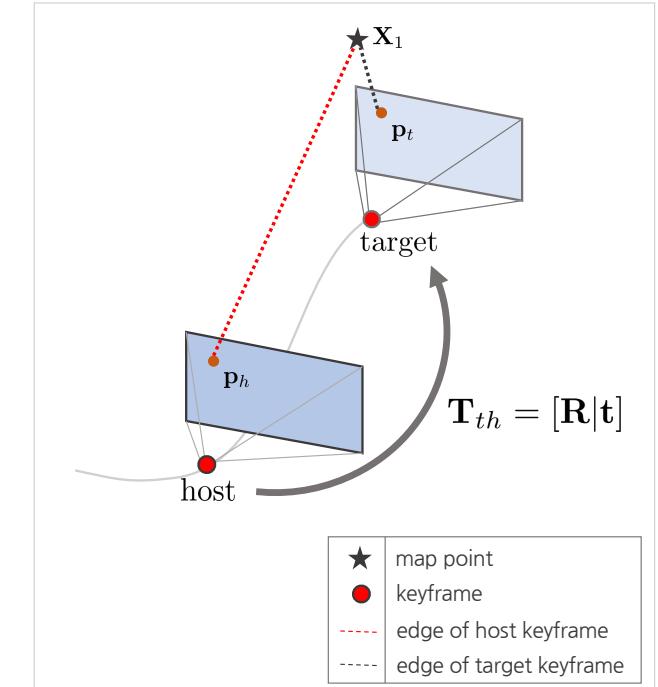
$$\frac{\partial \mathbf{r}}{\partial \mathbf{c}} = \frac{\partial \mathbf{r}}{\partial \mathbf{p}_t} \frac{\partial \mathbf{p}_t}{\partial \mathbf{c}}$$

현재까지 유도된 $\frac{\partial \mathbf{p}_t}{\partial \mathbf{c}}$ 는 다음과 같다.

$$\begin{aligned}\frac{\partial \mathbf{p}_t}{\partial \mathbf{c}} &= \begin{bmatrix} \frac{\partial u_t}{\partial f_x} & \frac{\partial u_t}{\partial f_y} & \frac{\partial u_t}{\partial c_x} & \frac{\partial u_t}{\partial c_y} \\ \frac{\partial v_t}{\partial f_x} & \frac{\partial v_t}{\partial f_y} & \frac{\partial v_t}{\partial c_x} & \frac{\partial v_t}{\partial c_y} \end{bmatrix} \\ &= \begin{bmatrix} \bar{u}_t + f_x \frac{\partial \bar{u}_t}{\partial f_x} & f_x \frac{\partial \bar{u}_t}{\partial f_y} & f_x \frac{\partial \bar{u}_t}{\partial c_x} + 1 & f_x \frac{\partial \bar{u}_t}{\partial c_y} \\ f_y \frac{\partial \bar{v}_t}{\partial f_x} & \bar{v}_t + f_y \frac{\partial \bar{v}_t}{\partial f_y} & f_y \frac{\partial \bar{v}_t}{\partial c_x} & f_y \frac{\partial \bar{v}_t}{\partial c_y} + 1 \end{bmatrix}\end{aligned}$$

$$\because u_t = f_x \bar{u}_t + c_x$$

$$v_t = f_y \bar{v}_t + c_y$$



4.3.2. Keyframes → Sliding Window Optimization → Jacobian Derivation of Camera Intrinsics

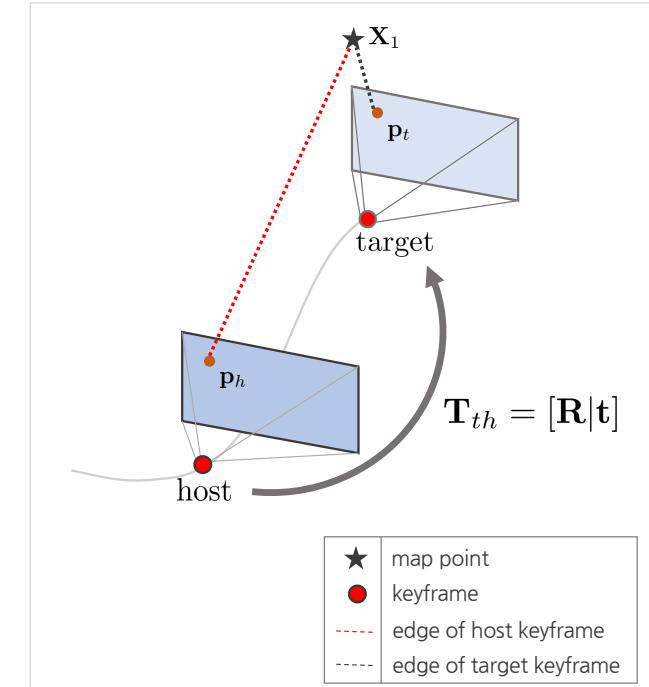
$$\frac{\partial \mathbf{r}}{\partial \mathbf{c}} = \frac{\partial \mathbf{r}}{\partial \mathbf{p}_t} \frac{\partial \mathbf{p}_t}{\partial \mathbf{c}}$$

현재까지 유도된 $\frac{\partial \mathbf{p}_t}{\partial \mathbf{c}}$ 는 다음과 같다.

$$\begin{aligned}\frac{\partial \mathbf{p}_t}{\partial \mathbf{c}} &= \begin{bmatrix} \frac{\partial u_t}{\partial f_x} & \frac{\partial u_t}{\partial f_y} & \frac{\partial u_t}{\partial c_x} & \frac{\partial u_t}{\partial c_y} \\ \frac{\partial v_t}{\partial f_x} & \frac{\partial v_t}{\partial f_y} & \frac{\partial v_t}{\partial c_x} & \frac{\partial v_t}{\partial c_y} \end{bmatrix} \\ &= \begin{bmatrix} \bar{u}_t + f_x \frac{\partial \bar{u}_t}{\partial f_x} & f_x \frac{\partial \bar{u}_t}{\partial f_y} & f_x \frac{\partial \bar{u}_t}{\partial c_x} + 1 & f_x \frac{\partial \bar{u}_t}{\partial c_y} \\ f_y \frac{\partial \bar{v}_t}{\partial f_x} & \bar{v}_t + f_y \frac{\partial \bar{v}_t}{\partial f_y} & f_y \frac{\partial \bar{v}_t}{\partial c_x} & f_y \frac{\partial \bar{v}_t}{\partial c_y} + 1 \end{bmatrix}\end{aligned}$$

$$\begin{aligned}\because u_t &= f_x \bar{u}_t + c_x \\ v_t &= f_y \bar{v}_t + c_y\end{aligned}$$

다음 순서로 $\begin{bmatrix} \frac{\partial \bar{u}_t}{\partial f_x} & \frac{\partial \bar{u}_t}{\partial f_y} & \frac{\partial \bar{u}_t}{\partial c_x} & \frac{\partial \bar{u}_t}{\partial c_y} \\ \frac{\partial \bar{v}_t}{\partial f_x} & \frac{\partial \bar{v}_t}{\partial f_y} & \frac{\partial \bar{v}_t}{\partial c_x} & \frac{\partial \bar{v}_t}{\partial c_y} \end{bmatrix}$ 값을 구해야 한다.



4.3.2. Keyframes → Sliding Window Optimization → Jacobian Derivation of Camera Intrinsics

$$\frac{\partial \mathbf{r}}{\partial \mathbf{c}} = \frac{\partial \mathbf{r}}{\partial \mathbf{p}_t} \frac{\partial \mathbf{p}_t}{\partial \mathbf{c}}$$

현재까지 유도된 $\frac{\partial \mathbf{p}_t}{\partial \mathbf{c}}$ 는 다음과 같다.

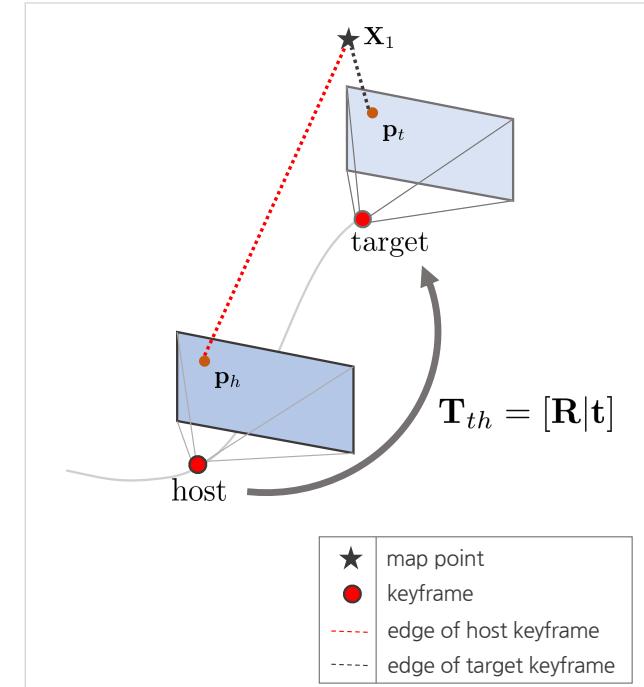
$$\begin{aligned}\frac{\partial \mathbf{p}_t}{\partial \mathbf{c}} &= \begin{bmatrix} \frac{\partial u_t}{\partial f_x} & \frac{\partial u_t}{\partial f_y} & \frac{\partial u_t}{\partial c_x} & \frac{\partial u_t}{\partial c_y} \\ \frac{\partial v_t}{\partial f_x} & \frac{\partial v_t}{\partial f_y} & \frac{\partial v_t}{\partial c_x} & \frac{\partial v_t}{\partial c_y} \end{bmatrix} \\ &= \begin{bmatrix} \bar{u}_t + f_x \frac{\partial \bar{u}_t}{\partial f_x} & f_x \frac{\partial \bar{u}_t}{\partial f_y} & f_x \frac{\partial \bar{u}_t}{\partial c_x} + 1 & f_x \frac{\partial \bar{u}_t}{\partial c_y} \\ f_y \frac{\partial \bar{v}_t}{\partial f_x} & \bar{v}_t + f_y \frac{\partial \bar{v}_t}{\partial f_y} & f_y \frac{\partial \bar{v}_t}{\partial c_x} & f_y \frac{\partial \bar{v}_t}{\partial c_y} + 1 \end{bmatrix}\end{aligned}$$

$$\begin{aligned}\because u_t &= f_x \bar{u}_t + c_x \\ v_t &= f_y \bar{v}_t + c_y\end{aligned}$$

다음 순서로 $\begin{bmatrix} \frac{\partial \bar{u}_t}{\partial f_x} & \frac{\partial \bar{u}_t}{\partial f_y} & \frac{\partial \bar{u}_t}{\partial c_x} & \frac{\partial \bar{u}_t}{\partial c_y} \\ \frac{\partial \bar{v}_t}{\partial f_x} & \frac{\partial \bar{v}_t}{\partial f_y} & \frac{\partial \bar{v}_t}{\partial c_x} & \frac{\partial \bar{v}_t}{\partial c_y} \end{bmatrix}$ 값을 구해야 한다.

우선 $[\bar{u}_t \quad \bar{v}_t \quad 1]$ 를 전개하면 다음과 같다.

$$\begin{aligned}[\bar{u}_t \quad \bar{v}_t \quad 1]^T &= \rho_t (\mathbf{R} \mathbf{X}_1 + \mathbf{t}) \\ &= \rho_t (\rho_h^{-1} \mathbf{R} \mathbf{K}^{-1} \mathbf{p}_h + \mathbf{t}) \\ &= \rho_t \rho_h^{-1} (\mathbf{R} \mathbf{K}^{-1} \mathbf{p}_h) + \rho_t \mathbf{t} \\ &= \rho_t \rho_h^{-1} \left[\begin{array}{c|c} \mathbf{R} & \end{array} \right] \left[\begin{array}{ccc} f_x^{-1} & -f_x^{-1} c_x \\ f_y^{-1} & -f_y^{-1} c_y \\ 1 & \end{array} \right] \left[\begin{array}{c} u_h \\ v_h \\ 1 \end{array} \right] + \rho_t \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} \\ &= \rho_t \rho_h^{-1} \left[\begin{array}{c|c} \mathbf{R} & \end{array} \right] \left[\begin{array}{c} f_x^{-1}(u_h - c_x) \\ f_y^{-1}(v_h - c_y) \\ 1 \end{array} \right] + \rho_t \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}\end{aligned}$$

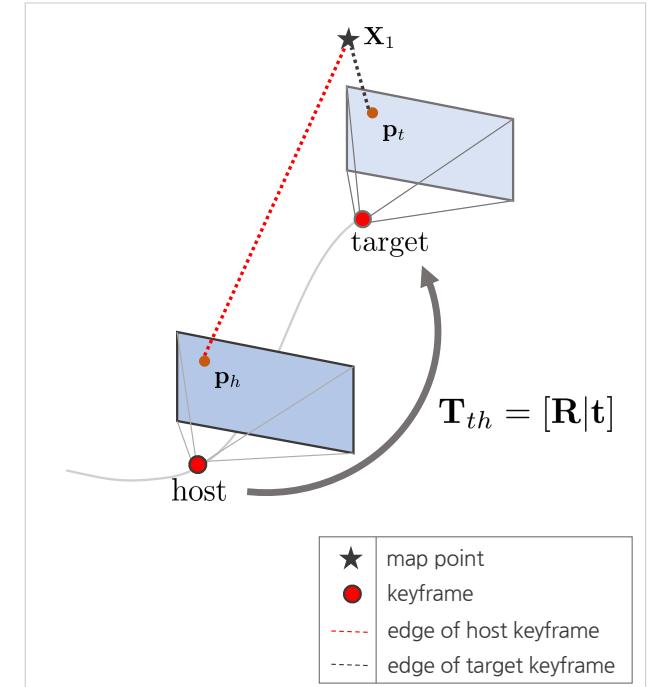


More detail: <https://blog.csdn.net/xxxinttp/article/details/90640350>

4.3.2. Keyframes → Sliding Window Optimization → Jacobian Derivation of Camera Intrinsics

$$\frac{\partial \mathbf{r}}{\partial \mathbf{c}} = \frac{\partial \mathbf{r}}{\partial \mathbf{p}_t} \frac{\partial \mathbf{p}_t}{\partial \mathbf{c}}$$

$$\begin{aligned}
 [\bar{u}_t & \quad \bar{v}_t & \quad 1]^T = \rho_t (\mathbf{R}\mathbf{X}_1 + \mathbf{t}) \\
 &= \rho_t (\rho_h^{-1} \mathbf{R} \mathbf{K}^{-1} \mathbf{p}_h + \mathbf{t}) \\
 &= \rho_t \rho_h^{-1} (\mathbf{R} \mathbf{K}^{-1} \mathbf{p}_h) + \rho_t \mathbf{t} \\
 &= \rho_t \rho_h^{-1} \begin{bmatrix} \mathbf{R} \end{bmatrix} \begin{bmatrix} f_x^{-1} & -f_x^{-1} c_x \\ f_y^{-1} & -f_y^{-1} c_y \\ 1 & \end{bmatrix} \begin{bmatrix} u_h \\ v_h \\ 1 \end{bmatrix} + \rho_t \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} \\
 &= \rho_t \rho_h^{-1} \begin{bmatrix} \mathbf{R} \end{bmatrix} \begin{bmatrix} f_x^{-1}(u_h - c_x) \\ f_y^{-1}(v_h - c_y) \\ 1 \end{bmatrix} + \rho_t \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} \\
 &= \rho_t \rho_h^{-1} \begin{bmatrix} r_{11} f_x^{-1}(u_h - c_x) + r_{12} f_y^{-1}(v_h - c_y) + r_{13} \\ r_{21} f_x^{-1}(u_h - c_x) + r_{22} f_y^{-1}(v_h - c_y) + r_{23} \\ r_{31} f_x^{-1}(u_h - c_x) + r_{32} f_y^{-1}(v_h - c_y) + r_{33} \end{bmatrix} + \rho_t \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}
 \end{aligned}$$



More detail: <https://blog.csdn.net/xxxinttp/article/details/90640350>

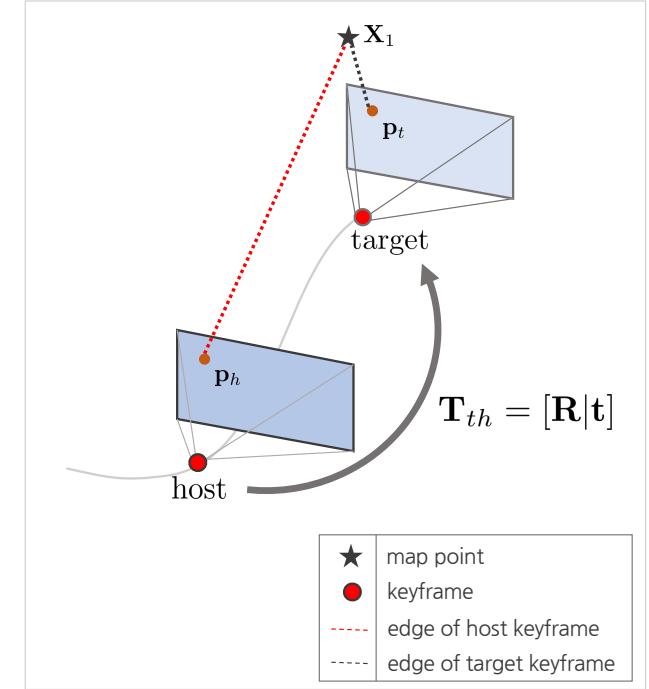
4.3.2. Keyframes → Sliding Window Optimization → Jacobian Derivation of Camera Intrinsics

$$\frac{\partial \mathbf{r}}{\partial \mathbf{c}} = \frac{\partial \mathbf{r}}{\partial \mathbf{p}_t} \frac{\partial \mathbf{p}_t}{\partial \mathbf{c}}$$

$$\begin{aligned}
 [\bar{u}_t & \quad \bar{v}_t & \quad 1]^T = \rho_t(\mathbf{R}\mathbf{X}_1 + \mathbf{t}) \\
 &= \rho_t(\rho_h^{-1}\mathbf{R}\mathbf{K}^{-1}\mathbf{p}_h + \mathbf{t}) \\
 &= \rho_t\rho_h^{-1}(\mathbf{R}\mathbf{K}^{-1}\mathbf{p}_h) + \rho_t\mathbf{t} \\
 &= \rho_t\rho_h^{-1} \begin{bmatrix} \mathbf{R} \end{bmatrix} \begin{bmatrix} f_x^{-1} & -f_x^{-1}c_x \\ f_y^{-1} & -f_y^{-1}c_y \\ 1 \end{bmatrix} \begin{bmatrix} u_h \\ v_h \\ 1 \end{bmatrix} + \rho_t \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} \\
 &= \rho_t\rho_h^{-1} \begin{bmatrix} \mathbf{R} \end{bmatrix} \begin{bmatrix} f_x^{-1}(u_h - c_x) \\ f_y^{-1}(v_h - c_y) \\ 1 \end{bmatrix} + \rho_t \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} \\
 &= \rho_t\rho_h^{-1} \begin{bmatrix} r_{11}f_x^{-1}(u_h - c_x) + r_{12}f_y^{-1}(v_h - c_y) + r_{13} \\ r_{21}f_x^{-1}(u_h - c_x) + r_{22}f_y^{-1}(v_h - c_y) + r_{23} \\ r_{31}f_x^{-1}(u_h - c_x) + r_{32}f_y^{-1}(v_h - c_y) + r_{33} \end{bmatrix} + \rho_t \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}
 \end{aligned}$$

따라서 \bar{u}_t, \bar{v}_t 는 다음과 같이 나타낼 수 있다.

$$\begin{aligned}
 \therefore \bar{u}_t &= \frac{r_{11}f_x^{-1}(u_h - c_x) + r_{12}f_y^{-1}(v_h - c_y) + r_{13} + \rho_h t_x}{r_{31}f_x^{-1}(u_h - c_x) + r_{32}f_y^{-1}(v_h - c_y) + r_{33} + \rho_h t_z} \\
 \bar{v}_t &= \frac{r_{21}f_x^{-1}(u_h - c_x) + r_{22}f_y^{-1}(v_h - c_y) + r_{23} + \rho_h t_y}{r_{31}f_x^{-1}(u_h - c_x) + r_{32}f_y^{-1}(v_h - c_y) + r_{33} + \rho_h t_z}
 \end{aligned}$$



4.3.2. Keyframes → Sliding Window Optimization → Jacobian Derivation of Camera Intrinsics

$$\frac{\partial \mathbf{r}}{\partial \mathbf{c}} = \frac{\partial \mathbf{r}}{\partial \mathbf{p}_t} \frac{\partial \mathbf{p}_t}{\partial \mathbf{c}}$$

$$\begin{aligned}\bar{u}_t &= \frac{r_{11}f_x^{-1}(u_h - c_x) + r_{12}f_y^{-1}(v_h - c_y) + r_{13} + \rho_h t_x}{r_{31}f_x^{-1}(u_h - c_x) + r_{32}f_y^{-1}(v_h - c_y) + r_{33} + \rho_h t_z} \\ \bar{v}_t &= \frac{r_{21}f_x^{-1}(u_h - c_x) + r_{22}f_y^{-1}(v_h - c_y) + r_{23} + \rho_h t_y}{r_{31}f_x^{-1}(u_h - c_x) + r_{32}f_y^{-1}(v_h - c_y) + r_{33} + \rho_h t_z}\end{aligned}$$

More detail: <https://blog.csdn.net/xxlhttp/article/details/90640350>

4.3.2. Keyframes → Sliding Window Optimization → Jacobian Derivation of Camera Intrinsics

$$\frac{\partial \mathbf{r}}{\partial \mathbf{c}} = \frac{\partial \mathbf{r}}{\partial \mathbf{p}_t} \frac{\partial \mathbf{p}_t}{\partial \mathbf{c}}$$

$$\begin{aligned}\bar{u}_t &= \frac{r_{11}f_x^{-1}(u_h - c_x) + r_{12}f_y^{-1}(v_h - c_y) + r_{13} + \rho_h t_x}{r_{31}f_x^{-1}(u_h - c_x) + r_{32}f_y^{-1}(v_h - c_y) + r_{33} + \rho_h t_z} \\ \bar{v}_t &= \frac{r_{21}f_x^{-1}(u_h - c_x) + r_{22}f_y^{-1}(v_h - c_y) + r_{23} + \rho_h t_y}{r_{31}f_x^{-1}(u_h - c_x) + r_{32}f_y^{-1}(v_h - c_y) + r_{33} + \rho_h t_z}\end{aligned}$$

위 식을 사용하여 $\begin{bmatrix} \frac{\partial \bar{u}_t}{\partial f_x} & \frac{\partial \bar{u}_t}{\partial f_y} & \frac{\partial \bar{u}_t}{\partial c_x} & \frac{\partial \bar{u}_t}{\partial c_y} \\ \frac{\partial \bar{v}_t}{\partial f_x} & \frac{\partial \bar{v}_t}{\partial f_y} & \frac{\partial \bar{v}_t}{\partial c_x} & \frac{\partial \bar{v}_t}{\partial c_y} \end{bmatrix}$ 를 구해보면 다음과 같다.

$$\begin{array}{ll} \frac{\partial \bar{u}_t}{\partial f_x} = \rho_t \rho_h^{-1} (r_{31} \bar{u}_t - r_{11}) f_x^{-2} (u_h - c_x) & \frac{\partial \bar{v}_t}{\partial f_x} = \rho_t \rho_h^{-1} (r_{31} \bar{v}_t - r_{21}) f_x^{-2} (u_h - c_x) \\ \frac{\partial \bar{u}_t}{\partial f_y} = \rho_t \rho_h^{-1} (r_{32} \bar{u}_t - r_{12}) f_y^{-2} (v_h - c_y) & \frac{\partial \bar{v}_t}{\partial f_y} = \rho_t \rho_h^{-1} (r_{32} \bar{v}_t - r_{22}) f_y^{-2} (v_h - c_y) \\ \frac{\partial \bar{u}_t}{\partial c_x} = \rho_t \rho_h^{-1} (r_{31} \bar{u}_t - r_{11}) f_x^{-1} & \frac{\partial \bar{v}_t}{\partial c_x} = \rho_t \rho_h^{-1} (r_{31} \bar{v}_t - r_{21}) f_x^{-1} \\ \frac{\partial \bar{u}_t}{\partial c_y} = \rho_t \rho_h^{-1} (r_{32} \bar{u}_t - r_{12}) f_y^{-1} & \frac{\partial \bar{v}_t}{\partial c_y} = \rho_t \rho_h^{-1} (r_{32} \bar{v}_t - r_{22}) f_y^{-1} \end{array}$$

4.3.2. Keyframes → Sliding Window Optimization → Jacobian Derivation of Camera Intrinsics

$$\frac{\partial \mathbf{r}}{\partial \mathbf{c}} = \frac{\partial \mathbf{r}}{\partial \mathbf{p}_t} \frac{\partial \mathbf{p}_t}{\partial \mathbf{c}}$$

최종적으로 $\frac{\partial \mathbf{p}_t}{\partial \mathbf{c}}$ 는 다음과 같이 나타낼 수 있다.

camera intrinsics(4)

$$\begin{aligned}\frac{\partial \mathbf{p}_t}{\partial \mathbf{c}} &= \begin{bmatrix} \frac{\partial u_t}{\partial f_x} & \frac{\partial u_t}{\partial f_y} & \frac{\partial u_t}{\partial c_x} & \frac{\partial u_t}{\partial c_y} \\ \frac{\partial v_t}{\partial f_x} & \frac{\partial v_t}{\partial f_y} & \frac{\partial v_t}{\partial c_x} & \frac{\partial v_t}{\partial c_y} \end{bmatrix} \\ &= \begin{bmatrix} \bar{u}_t + f_x \frac{\partial \bar{u}_t}{\partial f_x} & f_x \frac{\partial \bar{u}_t}{\partial f_y} & f_x \frac{\partial \bar{u}_t}{\partial c_x} + 1 & f_x \frac{\partial \bar{u}_t}{\partial c_y} \\ f_y \frac{\partial \bar{v}_t}{\partial f_x} & \bar{v}_t + f_y \frac{\partial \bar{v}_t}{\partial f_y} & f_y \frac{\partial \bar{v}_t}{\partial c_x} & f_y \frac{\partial \bar{v}_t}{\partial c_y} + 1 \end{bmatrix} \\ &= \begin{bmatrix} \bar{u}_t + \rho_t \rho_h^{-1} f_x^{-1} (r_{31} \bar{u}_t - r_{11}) (u_h - c_x) & \rho_t \rho_h^{-1} f_x f_y^{-2} (r_{32} \bar{u}_t - r_{12}) (v_h - c_y) & \rho_t \rho_h^{-1} (r_{31} \bar{u}_t - r_{11}) + 1 & \rho_t \rho_h^{-1} f_x f_y^{-1} (r_{32} \bar{u}_t - r_{12}) \\ \rho_t \rho_h^{-1} f_x^{-2} f_y (r_{31} \bar{v}_t - r_{21}) (u_h - c_x) & \bar{v}_t + \rho_t \rho_h^{-1} f_y^{-1} (r_{32} \bar{v}_t - r_{22}) (v_h - c_y) & \rho_t \rho_h^{-1} f_x^{-1} f_y (r_{31} \bar{v}_t - r_{21}) & \rho_t \rho_h^{-1} (r_{32} \bar{u}_t - r_{12}) + 1 \end{bmatrix}\end{aligned}$$

where,

$\frac{\partial \bar{u}_t}{\partial f_x} = \rho_t \rho_h^{-1} (r_{31} \bar{u}_t - r_{11}) f_x^{-2} (u_h - c_x)$	$\frac{\partial \bar{v}_t}{\partial f_x} = \rho_t \rho_h^{-1} (r_{31} \bar{v}_t - r_{21}) f_x^{-2} (u_h - c_x)$
$\frac{\partial \bar{u}_t}{\partial f_y} = \rho_t \rho_h^{-1} (r_{32} \bar{u}_t - r_{12}) f_y^{-2} (v_h - c_y)$	$\frac{\partial \bar{v}_t}{\partial f_y} = \rho_t \rho_h^{-1} (r_{32} \bar{v}_t - r_{22}) f_y^{-2} (v_h - c_y)$
$\frac{\partial \bar{u}_t}{\partial c_x} = \rho_t \rho_h^{-1} (r_{31} \bar{u}_t - r_{11}) f_x^{-1}$	$\frac{\partial \bar{v}_t}{\partial c_x} = \rho_t \rho_h^{-1} (r_{31} \bar{v}_t - r_{21}) f_x^{-1}$
$\frac{\partial \bar{u}_t}{\partial c_y} = \rho_t \rho_h^{-1} (r_{32} \bar{u}_t - r_{12}) f_y^{-1}$	$\frac{\partial \bar{u}_t}{\partial c_y} = \rho_t \rho_h^{-1} (r_{32} \bar{v}_t - r_{22}) f_y^{-1}$

4.3.3. Keyframes → Sliding Window Optimization → First Estimate Jacobian

Sliding Window Optimization을 수행하면 매 iteration마다 state variable \mathbf{x} 가 업데이트 된다.

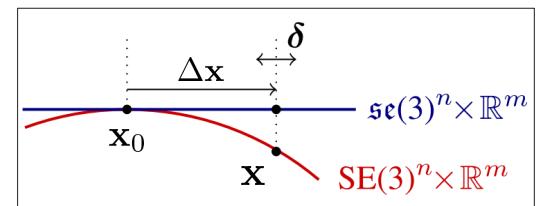
이 때 田 연산자는 $N+12$ 개의 상태를 각 파라미터 별로 업데이트해주는 연산자이다.

$$\mathbf{x} \leftarrow \Delta\mathbf{x} \boxplus \mathbf{x}$$

where, $\mathbf{x} = [\rho_1^{(1)}, \dots, \rho_1^{(N)}, \delta\xi^T, a, b, \mathbf{c}]_{(N+12) \times 1}^T$

Notation의 혼용을 방지하기 위해 논문의 기호를 일부 수정함

DSO paper Figure 6



\mathbf{x}_0	LBA를 시작하는 시점에서 state variable
\mathbf{x}	LBA가 끝난 후 state variable
$\Delta\mathbf{x}$	매 iteration 업데이트(marginalization 포함)로 인한 누적 증분량
δ	매 iteration 업데이트 증분량

4.3.3. Keyframes → Sliding Window Optimization → First Estimate Jacobian

Sliding Window Optimization을 수행하면 매 iteration마다 state variable \mathbf{x} 가 업데이트 된다.

이 때 田 연산자는 $N+12$ 개의 상태를 각 파라미터 별로 업데이트해주는 연산자이다.

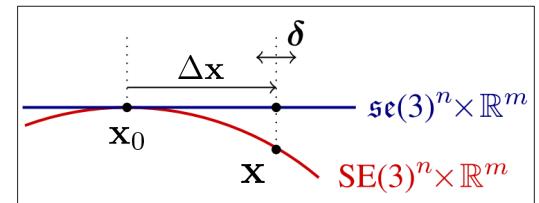
$$\mathbf{x} \leftarrow \Delta\mathbf{x} \boxplus \mathbf{x}$$

where, $\mathbf{x} = [\rho_1^{(1)}, \dots, \rho_1^{(N)}, \delta\xi^T, a, b, \mathbf{c}]_{(N+12) \times 1}^T$

DSO에서는 LBA를 수행할 때마다 6번의 최적화를 수행하는데 만약 매 iteration마다 Jacobian을 새로 계산하면 다음과 같은 단점이 존재한다. (자세한 내용은 우측 하단의 링크 참조)

Notation의 혼용을 방지하기 위해 논문의 기호를 일부 수정함

DSO paper Figure 6



x_0	LBA를 시작하는 시점에서 state variable
x	LBA가 끝난 후 state variable
Δx	매 iteration 업데이트(marginalization 포함)로 인한 누적 증분량
δ	매 iteration 업데이트 증분량

4.3.3. Keyframes → Sliding Window Optimization → First Estimate Jacobian

Sliding Window Optimization을 수행하면 매 iteration마다 state variable \mathbf{x} 가 업데이트 된다.

이 때 田 연산자는 $N+12$ 개의 상태를 각 파라미터 별로 업데이트해주는 연산자이다.

$$\mathbf{x} \leftarrow \Delta\mathbf{x} \boxplus \mathbf{x}$$

where, $\mathbf{x} = [\rho_1^{(1)}, \dots, \rho_1^{(N)}, \delta\xi^T, a, b, \mathbf{c}]_{(N+12) \times 1}^T$

DSO에서는 LBA를 수행할 때마다 6번의 최적화를 수행하는데 만약 매 iteration마다 Jacobian을 새로 계산하면 다음과 같은 단점이 존재한다. (자세한 내용은 우측 하단의 링크 참조)

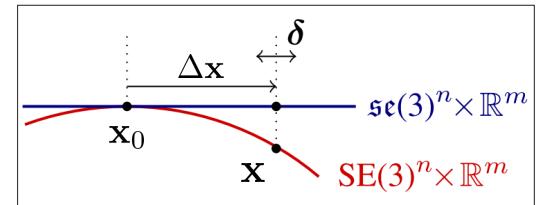
1) 연산량이 매우 증가하므로 실시간 성능을 보장할 수 없다.

2) 서로 다른 state의 Jacobian을 한 시스템에서 같이 사용하는 경우 성능의 저하를 일으킬 가능성이 존재한다.

Adding linearizations around different evaluation points eliminates these and thus slowly corrupts the system. (DSO paper)

Notation의 혼용을 방지하기 위해 논문의 기호를 일부 수정함

DSO paper Figure 6



\mathbf{x}_0	LBA를 시작하는 시점에서 state variable
\mathbf{x}	LBA가 끝난 후 state variable
$\Delta\mathbf{x}$	매 iteration 업데이트(marginalization 포함)로 인한 누적 증분량
δ	매 iteration 업데이트 증분량

4.3.3. Keyframes → Sliding Window Optimization → First Estimate Jacobian

Sliding Window Optimization을 수행하면 매 iteration마다 state variable \mathbf{x} 가 업데이트 된다.

이 때 田 연산자는 $N+12$ 개의 상태를 각 파라미터 별로 업데이트해주는 연산자이다.

$$\mathbf{x} \leftarrow \Delta\mathbf{x} \boxplus \mathbf{x}$$

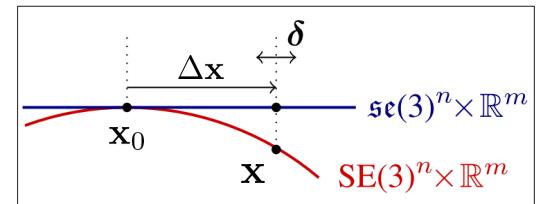
where, $\mathbf{x} = [\rho_1^{(1)}, \dots, \rho_1^{(N)}, \delta\xi^T, a, b, \mathbf{c}]_{(N+12) \times 1}^T$

DSO에서는 LBA를 수행할 때마다 6번의 최적화를 수행하는데 만약 매 iteration마다 Jacobian을 새로 계산하면 다음과 같은 단점이 존재한다. (자세한 내용은 우측 하단의 링크 참조)

- 1) 연산량이 매우 증가하므로 실시간 성능을 보장할 수 없다.
- 2) 서로 다른 state의 Jacobian을 한 시스템에서 같이 사용하는 경우 성능의 저하를 일으킬 가능성이 존재한다.

Notation의 혼용을 방지하기 위해 논문의 기호를 일부 수정함

DSO paper Figure 6



x_0	LBA를 시작하는 시점에서 state variable
x	LBA가 끝난 후 state variable
Δx	매 iteration 업데이트(marginalization 포함)로 인한 누적 증분량
δ	매 iteration 업데이트 증분량

따라서 LBA를 수행할 때 매 iteration마다(marginalization 포함) 업데이트로 인해 state variable \mathbf{x} 가 변하더라도

Jacobian은 새로 계산하지 않고 LBA를 시작하는 시점(x_0)에서 계산한 Jacobian을 사용하여 업데이트하는 방법을 First Estimate Jacobian(FEJ)라고 한다.

FEJ의 적용 순서는 다음과 같다.

4.3.3. Keyframes → Sliding Window Optimization → First Estimate Jacobian

Sliding Window Optimization을 수행하면 매 iteration마다 state variable \mathbf{x} 가 업데이트 된다.

이 때 田 연산자는 $N+12$ 개의 상태를 각 파라미터 별로 업데이트해주는 연산자이다.

$$\mathbf{x} \leftarrow \Delta\mathbf{x} \boxplus \mathbf{x}$$

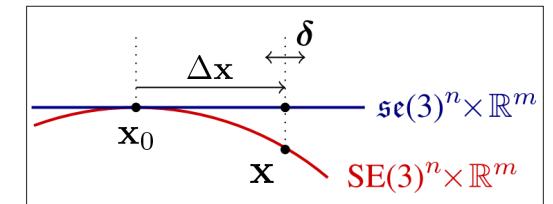
where, $\mathbf{x} = [\rho_1^{(1)}, \dots, \rho_1^{(N)}, \delta\xi^T, a, b, \mathbf{c}]_{(N+12) \times 1}^T$

DSO에서는 LBA를 수행할 때마다 6번의 최적화를 수행하는데 만약 매 iteration마다 Jacobian을 새로 계산하면 다음과 같은 단점이 존재한다. (자세한 내용은 우측 하단의 링크 참조)

- 1) 연산량이 매우 증가하므로 실시간 성능을 보장할 수 없다.
- 2) 서로 다른 state의 Jacobian을 한 시스템에서 같이 사용하는 경우 성능의 저하를 일으킬 가능성이 존재한다.

Notation의 혼용을 방지하기 위해 논문의 기호를 일부 수정함

DSO paper Figure 6



\mathbf{x}_0	LBA를 시작하는 시점에서 state variable
\mathbf{x}	LBA가 끝난 후 state variable
$\Delta\mathbf{x}$	매 iteration 업데이트(marginalization 포함)로 인한 누적 증분량
δ	매 iteration 업데이트 증분량

따라서 LBA를 수행할 때 매 iteration마다(marginalization 포함) 업데이트로 인해 state variable \mathbf{x} 가 변하더라도 Jacobian은 새로 계산하지 않고 LBA를 시작하는 시점(\mathbf{x}_0)에서 계산한 Jacobian을 사용하여 업데이트하는 방법을 First Estimate Jacobian(FEJ)라고 한다. FEJ의 적용 순서는 다음과 같다.

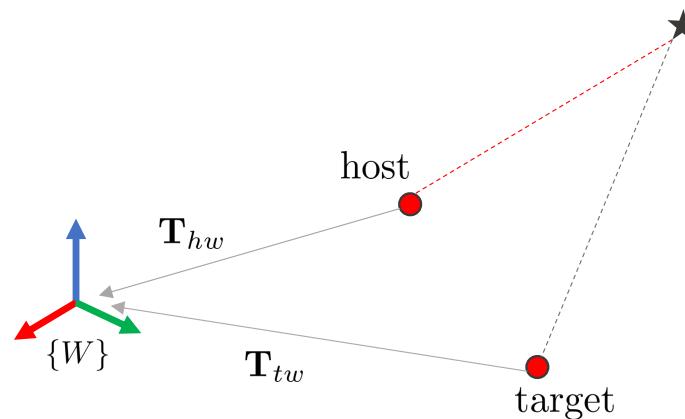
- 1) LBA를 시작하기 전 현재 \mathbf{x} 를 \mathbf{x}_0 에 저장한 후 Jacobian을 계산한다. $\mathbf{x}_0 \leftarrow \mathbf{x}$, compute $\mathbf{J}|_{\mathbf{x}_0}$
- 2) LBA를 수행하면서 매 iteration마다 업데이트 값을 누적시킨다. $\Delta\mathbf{x} \leftarrow \delta + \Delta\mathbf{x}$
- 3) 매 iteration동안 Jacobian은 \mathbf{x}_0 에서 계산한 값을 사용한다.
- 4) LBA iteration이 끝난 경우 현재 상태에서 누적된 증분량을 업데이트한다. $\mathbf{x} \leftarrow \Delta\mathbf{x} \boxplus \mathbf{x}_0$

more detail: <https://blog.csdn.net/heyijia0327/article/details/53707261>

4.3.4. Keyframes → Sliding Window Optimization → Adjoint Transformation

앞서 설명한 것과 같이 DSO에서는 하나의 맵포인트에 2개의 키프레임(host, target)이 연결되어 있다.

이에 따라 최적화 수행 시 연산량을 줄이기 위해 target, host 키프레임 사이의 상대 포즈를 기반으로 계산한 Jacobian $\frac{\partial \mathbf{r}}{\partial \xi_{th}}$ 를 사용한다.



★	map point
●	keyframe
\mathbf{T}_{hw}	host 키프레임에서 바라본 월드좌표계 {w}의 포즈(4×4 행렬)
\mathbf{T}_{tw}	target 키프레임에서 바라본 월드좌표계 {w}의 포즈(4×4 행렬)
\mathbf{r}	residual
ξ_{th}	target 기준 host 키프레임의 상대 포즈 변화량(twist)(6차원벡터)

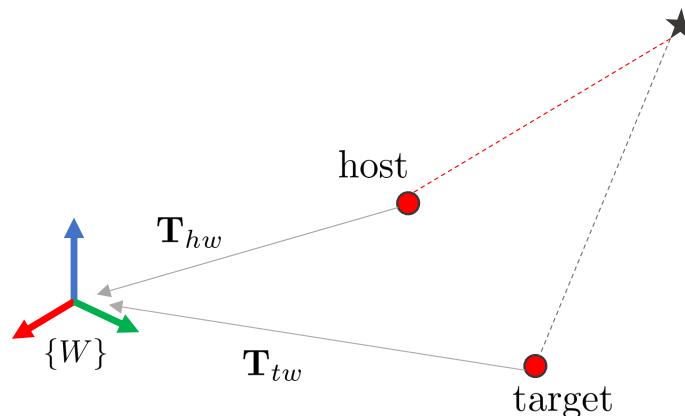
more detail: <https://www.cnblogs.com/JingeTU/p/8306727.html>

4.3.4. Keyframes → Sliding Window Optimization → Adjoint Transformation

앞서 설명한 것과 같이 DSO에서는 하나의 맵포인트에 2개의 키프레임(host, target)이 연결되어 있다.

이에 따라 최적화 수행 시 연산량을 줄이기 위해 target, host 키프레임 사이의 상대 포즈를 기반으로 계산한 Jacobian $\frac{\partial \mathbf{r}}{\partial \xi_{th}}$ 를 사용한다.

하지만 해당 Jacobian을 사용하여 Iterative Solution을 구하게 되면 두 host, target 키프레임 사이의 상대포즈만 최적화되므로 월드좌표계 $\{W\}$ 를 기준으로 한 카메라 포즈는 계산할 수 없다.



★	map point
●	keyframe
\mathbf{T}_{hw}	host 키프레임에서 바라본 월드좌표계 $\{w\}$ 의 포즈(4×4 행렬)
\mathbf{T}_{tw}	target 키프레임에서 바라본 월드좌표계 $\{w\}$ 의 포즈(4×4 행렬)
\mathbf{r}	residual
ξ_{th}	target 기준 host 키프레임의 상대 포즈 변화량(twist)(6차원벡터)

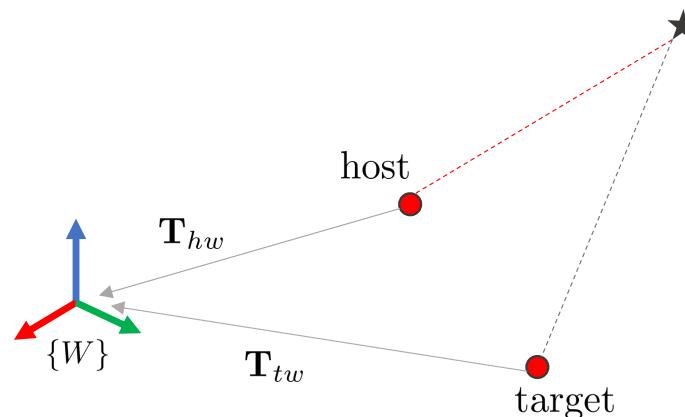
4.3.4. Keyframes → Sliding Window Optimization → Adjoint Transformation

앞서 설명한 것과 같이 DSO에서는 하나의 맵포인트에 2개의 키프레임(host, target)이 연결되어 있다.

이에 따라 최적화 수행 시 연산량을 줄이기 위해 target, host 키프레임 사이의 상대 포즈를 기반으로 계산한 Jacobian $\frac{\partial \mathbf{r}}{\partial \xi_{th}}$ 를 사용한다.

하지만 해당 Jacobian을 사용하여 Iterative Solution을 구하게 되면 두 host, target 키프레임 사이의 상대포즈만 최적화되므로 월드좌표계 $\{W\}$ 를 기준으로 한 카메라 포즈는 계산할 수 없다.

이 때, DSO에서는 **Adjoint Transformation**을 통해 $\frac{\partial \mathbf{r}}{\partial \xi_{hw}}$, $\frac{\partial \mathbf{r}}{\partial \xi_{tw}}$ 를 계산하여 월드좌표계의 카메라 포즈를 최적화하였다.



★	map point
●	keyframe
\mathbf{T}_{hw}	host 키프레임에서 바라본 월드좌표계 $\{w\}$ 의 포즈(4×4 행렬)
\mathbf{T}_{tw}	target 키프레임에서 바라본 월드좌표계 $\{w\}$ 의 포즈(4×4 행렬)
\mathbf{r}	residual
ξ_{th}	target 기준 host 키프레임의 상대 포즈 변화량(twist)(6차원벡터)

4.3.4. Keyframes → Sliding Window Optimization → Adjoint Transformation

$\frac{\partial \mathbf{r}}{\partial \xi_{th}}$ 을 $\frac{\partial \mathbf{r}}{\partial \xi_{hw}}$, $\frac{\partial \mathbf{r}}{\partial \xi_{tw}}$ 로 변환하기 위해 다음과 같은 연산을 사용한다.

host

$$\begin{aligned} \frac{\partial \mathbf{r}}{\partial \xi_{hw}}^T \frac{\partial \mathbf{r}}{\partial \xi_{hw}} &= \left(\frac{\partial \mathbf{r}}{\partial \xi_{th}} \frac{\partial \xi_{th}}{\partial \xi_{hw}} \right)^T \left(\frac{\partial \mathbf{r}}{\partial \xi_{th}} \frac{\partial \xi_{th}}{\partial \xi_{hw}} \right) \\ &= \left(\frac{\partial \xi_{th}}{\partial \xi_{hw}} \right)^T \left(\frac{\partial \mathbf{r}}{\partial \xi_{th}} \right)^T \left(\frac{\partial \mathbf{r}}{\partial \xi_{th}} \right) \left(\frac{\partial \xi_{th}}{\partial \xi_{hw}} \right) \end{aligned}$$

target

$$\begin{aligned} \frac{\partial \mathbf{r}}{\partial \xi_{tw}}^T \frac{\partial \mathbf{r}}{\partial \xi_{tw}} &= \left(\frac{\partial \mathbf{r}}{\partial \xi_{th}} \frac{\partial \xi_{th}}{\partial \xi_{tw}} \right)^T \left(\frac{\partial \mathbf{r}}{\partial \xi_{th}} \frac{\partial \xi_{th}}{\partial \xi_{tw}} \right) \\ &= \left(\frac{\partial \xi_{th}}{\partial \xi_{tw}} \right)^T \left(\frac{\partial \mathbf{r}}{\partial \xi_{th}} \right)^T \left(\frac{\partial \mathbf{r}}{\partial \xi_{th}} \right) \left(\frac{\partial \xi_{th}}{\partial \xi_{tw}} \right) \end{aligned}$$

\mathbf{r}	residual
ξ_{th}	target 기준 host 키프레임의 상대 포즈 변화량(twist)(6차원벡터)
ξ_{hw}	전역좌표계에서 본 host 키프레임의 상대 포즈 변화량(twist)(6차원벡터)
ξ_{tw}	전역좌표계에서 본 target 키프레임의 상대 포즈 변화량(twist)(6차원벡터)

more detail: <https://www.cnblogs.com/JingeTU/p/8306727.html>

4.3.4. Keyframes → Sliding Window Optimization → Adjoint Transformation

$\frac{\partial \mathbf{r}}{\partial \xi_{th}}$ 을 $\frac{\partial \mathbf{r}}{\partial \xi_{hw}}$, $\frac{\partial \mathbf{r}}{\partial \xi_{tw}}$ 로 변환하기 위해 다음과 같은 연산을 사용한다.

host	target
$\begin{aligned} \frac{\partial \mathbf{r}}{\partial \xi_{hw}}^T \frac{\partial \mathbf{r}}{\partial \xi_{hw}} &= \left(\frac{\partial \mathbf{r}}{\partial \xi_{th}} \frac{\partial \xi_{th}}{\partial \xi_{hw}} \right)^T \left(\frac{\partial \mathbf{r}}{\partial \xi_{th}} \frac{\partial \xi_{th}}{\partial \xi_{hw}} \right) \\ &= \left(\frac{\partial \xi_{th}}{\partial \xi_{hw}} \right)^T \left(\frac{\partial \mathbf{r}}{\partial \xi_{th}} \right)^T \left(\frac{\partial \mathbf{r}}{\partial \xi_{th}} \right) \left(\frac{\partial \xi_{th}}{\partial \xi_{hw}} \right) \end{aligned}$	$\begin{aligned} \frac{\partial \mathbf{r}}{\partial \xi_{tw}}^T \frac{\partial \mathbf{r}}{\partial \xi_{tw}} &= \left(\frac{\partial \mathbf{r}}{\partial \xi_{th}} \frac{\partial \xi_{th}}{\partial \xi_{tw}} \right)^T \left(\frac{\partial \mathbf{r}}{\partial \xi_{th}} \frac{\partial \xi_{th}}{\partial \xi_{tw}} \right) \\ &= \left(\frac{\partial \xi_{th}}{\partial \xi_{tw}} \right)^T \left(\frac{\partial \mathbf{r}}{\partial \xi_{th}} \right)^T \left(\frac{\partial \mathbf{r}}{\partial \xi_{th}} \right) \left(\frac{\partial \xi_{th}}{\partial \xi_{tw}} \right) \end{aligned}$

이 때 twist 변화량은 다음과 같이 계산할 수 있다.

$\frac{\partial \xi_{th}}{\partial \xi_{hw}} = -\text{Ad}_{\mathbf{T}_{th}}$	$\frac{\partial \xi_{th}}{\partial \xi_{tw}} = \mathbf{I}$
------------------------------------------------------------------------------	------------------------------------------------------------

r	residual
ξ_{th}	target 기준 host 키프레임의 상대 포즈 변화량(twist)(6차원벡터)
ξ_{hw}	전역좌표계에서 본 host 키프레임의 상대 포즈 변화량(twist)(6차원벡터)
ξ_{tw}	전역좌표계에서 본 target 키프레임의 상대 포즈 변화량(twist)(6차원벡터)

more detail: <https://www.cnblogs.com/JingeTU/p/8306727.html>

4.3.4. Keyframes → Sliding Window Optimization → Adjoint Transformation

$\frac{\partial \mathbf{r}}{\partial \xi_{th}}$ 을 $\frac{\partial \mathbf{r}}{\partial \xi_{hw}}$, $\frac{\partial \mathbf{r}}{\partial \xi_{tw}}$ 로 변환하기 위해 다음과 같은 연산을 사용한다.

host	target
$\begin{aligned} \frac{\partial \mathbf{r}}{\partial \xi_{hw}}^T \frac{\partial \mathbf{r}}{\partial \xi_{hw}} &= \left(\frac{\partial \mathbf{r}}{\partial \xi_{th}} \frac{\partial \xi_{th}}{\partial \xi_{hw}} \right)^T \left(\frac{\partial \mathbf{r}}{\partial \xi_{th}} \frac{\partial \xi_{th}}{\partial \xi_{hw}} \right) \\ &= \left(\frac{\partial \xi_{th}}{\partial \xi_{hw}} \right)^T \left(\frac{\partial \mathbf{r}}{\partial \xi_{th}} \right)^T \left(\frac{\partial \mathbf{r}}{\partial \xi_{th}} \right) \left(\frac{\partial \xi_{th}}{\partial \xi_{hw}} \right) \end{aligned}$	$\begin{aligned} \frac{\partial \mathbf{r}}{\partial \xi_{tw}}^T \frac{\partial \mathbf{r}}{\partial \xi_{tw}} &= \left(\frac{\partial \mathbf{r}}{\partial \xi_{th}} \frac{\partial \xi_{th}}{\partial \xi_{tw}} \right)^T \left(\frac{\partial \mathbf{r}}{\partial \xi_{th}} \frac{\partial \xi_{th}}{\partial \xi_{tw}} \right) \\ &= \left(\frac{\partial \xi_{th}}{\partial \xi_{tw}} \right)^T \left(\frac{\partial \mathbf{r}}{\partial \xi_{th}} \right)^T \left(\frac{\partial \mathbf{r}}{\partial \xi_{th}} \right) \left(\frac{\partial \xi_{th}}{\partial \xi_{tw}} \right) \end{aligned}$

이 때 twist 변화량은 다음과 같이 계산할 수 있다.

$\frac{\partial \xi_{th}}{\partial \xi_{hw}} = -\text{Ad}_{\mathbf{T}_{th}}$	$\frac{\partial \xi_{th}}{\partial \xi_{tw}} = \mathbf{I}$
------------------------------------------------------------------------------	------------------------------------------------------------

여기서 $\text{Ad}_{\mathbf{T}_{th}}$ 를 Adjoint Transformation of SE(3) Group라고 한다.

해당 변환이 하는 역할은 3차원 공간 상에서 특정 좌표계(또는 포즈)에서 바라본 twist를 다른 좌표계(또는 포즈)에서 바라본 twist로 변환하는 역할을 한다.

$$\xi_{tw} = \text{Ad}_{\mathbf{T}_{th}} \cdot \xi_{hw}$$

\mathbf{r}	residual
ξ_{th}	target 기준 host 키프레임의 상대 포즈 변화량(twist)(6차원벡터)
ξ_{hw}	전역좌표계에서 본 host 키프레임의 상대 포즈 변화량(twist)(6차원벡터)
ξ_{tw}	전역좌표계에서 본 target 키프레임의 상대 포즈 변화량(twist)(6차원벡터)

more detail: <https://www.cnblogs.com/JingeTU/p/8306727.html>

4.3.4. Keyframes → Sliding Window Optimization → Adjoint Transformation

$\frac{\partial \mathbf{r}}{\partial \xi_{th}}$ 을 $\frac{\partial \mathbf{r}}{\partial \xi_{hw}}$, $\frac{\partial \mathbf{r}}{\partial \xi_{tw}}$ 로 변환하기 위해 다음과 같은 연산을 사용한다.

host	target
$\begin{aligned} \frac{\partial \mathbf{r}}{\partial \xi_{hw}}^T \frac{\partial \mathbf{r}}{\partial \xi_{hw}} &= \left(\frac{\partial \mathbf{r}}{\partial \xi_{th}} \frac{\partial \xi_{th}}{\partial \xi_{hw}} \right)^T \left(\frac{\partial \mathbf{r}}{\partial \xi_{th}} \frac{\partial \xi_{th}}{\partial \xi_{hw}} \right) \\ &= \left(\frac{\partial \xi_{th}}{\partial \xi_{hw}} \right)^T \left(\frac{\partial \mathbf{r}}{\partial \xi_{th}} \right)^T \left(\frac{\partial \mathbf{r}}{\partial \xi_{th}} \right) \left(\frac{\partial \xi_{th}}{\partial \xi_{hw}} \right) \end{aligned}$	$\begin{aligned} \frac{\partial \mathbf{r}}{\partial \xi_{tw}}^T \frac{\partial \mathbf{r}}{\partial \xi_{tw}} &= \left(\frac{\partial \mathbf{r}}{\partial \xi_{th}} \frac{\partial \xi_{th}}{\partial \xi_{tw}} \right)^T \left(\frac{\partial \mathbf{r}}{\partial \xi_{th}} \frac{\partial \xi_{th}}{\partial \xi_{tw}} \right) \\ &= \left(\frac{\partial \xi_{th}}{\partial \xi_{tw}} \right)^T \left(\frac{\partial \mathbf{r}}{\partial \xi_{th}} \right)^T \left(\frac{\partial \mathbf{r}}{\partial \xi_{th}} \right) \left(\frac{\partial \xi_{th}}{\partial \xi_{tw}} \right) \end{aligned}$

이 때 twist 변화량은 다음과 같이 계산할 수 있다.

$\frac{\partial \xi_{th}}{\partial \xi_{hw}} = -\text{Ad}_{\mathbf{T}_{th}}$	$\frac{\partial \xi_{th}}{\partial \xi_{tw}} = \mathbf{I}$
------------------------------------------------------------------------------	------------------------------------------------------------

여기서 $\text{Ad}_{\mathbf{T}_{th}}$ 를 Adjoint Transformation of SE(3) Group라고 한다.

해당 변환이 하는 역할은 3차원 공간 상에서 특정 좌표계(또는 포즈)에서 바라본 twist를 다른 좌표계(또는 포즈)에서 바라본 twist로 변환하는 역할을 한다.

$$\xi_{tw} = \text{Ad}_{\mathbf{T}_{th}} \cdot \xi_{hw}$$

구체적으로 $\text{Ad}_{\mathbf{T}_{th}}$ 는 host 키프레임에서 바라본 특정 twist를 target 키프레임에서 바라본 twist로 변환하는 연산자이다.

하지만 해당 슬라이드에서는 $\frac{\partial \xi_{th}}{\partial \xi_{hw}}$ 값으로 유도되었기 때문에 $\frac{\partial \mathbf{r}}{\partial \xi_{hw}}$ 를 계산하기 위한 도구로만 사용된다.

\mathbf{r}	residual
ξ_{th}	target 기준 host 키프레임의 상대 포즈 변화량(twist)(6차원벡터)
ξ_{hw}	전역좌표계에서 본 host 키프레임의 상대 포즈 변화량(twist)(6차원벡터)
ξ_{tw}	전역좌표계에서 본 target 키프레임의 상대 포즈 변화량(twist)(6차원벡터)

more detail: <https://www.cnblogs.com/JingeTU/p/8306727.html>

4.3.4. Keyframes → Sliding Window Optimization → Adjoint Transformation

$$\frac{\partial \xi_{th}}{\partial \xi_{hw}} = -\text{Ad}_{\mathbf{T}_{th}}$$

$$\frac{\partial \xi_{th}}{\partial \xi_{tw}} = \mathbf{I}$$

$\text{Ad}_{\mathbf{T}_{th}}$ 의 자세한 정의 및 역할에 대해 자세히 알고 싶으면 우측 하단의 링크를 참조하면 된다.

$$\exp(\text{Ad}_{\mathbf{T}_{th}} \cdot \xi^\wedge) = \mathbf{T}_{th} \exp(\xi^\wedge) \mathbf{T}_{th}^{-1}$$

$$\delta \xi^\wedge = \begin{bmatrix} \delta \mathbf{w}^\wedge & \delta \mathbf{v} \\ \mathbf{0}^T & 0 \end{bmatrix} \in \mathbb{R}^{4 \times 4}$$

$$\delta \xi = \begin{bmatrix} \delta \mathbf{v} \\ \delta \mathbf{w} \end{bmatrix} = \begin{bmatrix} \delta v_x \\ \delta v_y \\ \delta v_z \\ \delta w_x \\ \delta w_y \\ \delta w_z \end{bmatrix} \in \mathbb{R}^6$$

$\delta \xi \in \mathbb{R}^6$	상대 포즈 변화량 (twist)(6차원벡터)
$\delta \xi^\wedge \in \text{se}(3)$	hat 연산자가 적용된 twist (4x4행렬)(lie algebra)
$\exp(\xi^\wedge) \in \text{SE}(3)$	3차원 포즈 (transform)(4x4행렬)(lie group)

Lie Theory for Robotics 관련 내용은 다음 자료 참조: <http://ethaneade.com/lie.pdf>

more detail: <https://www.cnblogs.com/JingeTU/p/8306727.html>

4.3.4. Keyframes → Sliding Window Optimization → Adjoint Transformation

$$\frac{\partial \xi_{th}}{\partial \xi_{hw}} = -\text{Ad}_{\mathbf{T}_{th}}$$

$$\frac{\partial \xi_{th}}{\partial \xi_{tw}} = \mathbf{I}$$

$\text{Ad}_{\mathbf{T}_{th}}$ 의 자세한 정의 및 역할에 대해 자세히 알고 싶으면 우측 하단의 링크를 참조하면 된다.

$$\exp(\text{Ad}_{\mathbf{T}_{th}} \cdot \xi^\wedge) = \mathbf{T}_{th} \exp(\xi^\wedge) \mathbf{T}_{th}^{-1}$$

해당 슬라이드에서는 $\frac{\partial \xi_{th}}{\partial \xi_{hw}}$ $\frac{\partial \xi_{th}}{\partial \xi_{tw}}$ 두 값의 유도 과정에 대해 설명한다.

우선 $\mathbf{T}_{th} = \mathbf{T}_{tw} \mathbf{T}_{hw}^{-1}$ 과 같이 2개의 포즈로 분할할 수 있고 이를 Iterative 업데이트 공식으로 확장하면 다음과 같다.

$$\delta \xi^\wedge = \begin{bmatrix} \delta \mathbf{w}^\wedge & \delta \mathbf{v} \\ \mathbf{0}^T & 0 \end{bmatrix} \in \mathbb{R}^{4 \times 4}$$

$$\delta \xi = \begin{bmatrix} \delta \mathbf{v} \\ \delta \mathbf{w} \end{bmatrix} = \begin{bmatrix} \delta v_x \\ \delta v_y \\ \delta v_z \\ \delta w_x \\ \delta w_y \\ \delta w_z \end{bmatrix} \in \mathbb{R}^6$$

$\delta \xi \in \mathbb{R}^6$	상대 포즈 변화량 (twist)(6차원벡터)
$\delta \xi^\wedge \in \text{se}(3)$	hat 연산자가 적용된 twist (4x4행렬)(lie algebra)
$\exp(\xi^\wedge) \in \text{SE}(3)$	3차원 포즈 (transform)(4x4행렬)(lie group)

Lie Theory for Robotics 관련 내용은 다음 자료 참조: <http://ethaneade.com/lie.pdf>

more detail: <https://www.cnblogs.com/JingeTU/p/8306727.html>

4.3.4. Keyframes → Sliding Window Optimization → Adjoint Transformation

$$\frac{\partial \xi_{th}}{\partial \xi_{hw}} = -\text{Ad}_{\mathbf{T}_{th}}$$

$$\frac{\partial \xi_{th}}{\partial \xi_{tw}} = \mathbf{I}$$

$\text{Ad}_{\mathbf{T}_{th}}$ 의 자세한 정의 및 역할에 대해 자세히 알고 싶으면 우측 하단의 링크를 참조하면 된다.

$$\exp(\text{Ad}_{\mathbf{T}_{th}} \cdot \xi^\wedge) = \mathbf{T}_{th} \exp(\xi^\wedge) \mathbf{T}_{th}^{-1}$$

해당 슬라이드에서는 $\frac{\partial \xi_{th}}{\partial \xi_{hw}}$ $\frac{\partial \xi_{th}}{\partial \xi_{tw}}$ 두 값의 유도 과정에 대해 설명한다.

우선 $\mathbf{T}_{th} = \mathbf{T}_{tw} \mathbf{T}_{hw}^{-1}$ 과 같이 2개의 포즈로 분할할 수 있고 이를 Iterative 업데이트 공식으로 확장하면 다음과 같다.

host

$$\begin{aligned} \exp(\xi_{th}^\wedge + \delta\xi_{th}^\wedge) \mathbf{T}_{th} &= \mathbf{T}_{tw} (\exp(\xi_{hw}^\wedge + \delta\xi_{hw}^\wedge) \mathbf{T}_{hw})^{-1} \\ \exp(\xi_{th}^\wedge + \delta\xi_{th}^\wedge) &= \mathbf{T}_{tw} \mathbf{T}_{hw}^{-1} \exp(-(\xi_{hw}^\wedge + \delta\xi_{hw}^\wedge)) \mathbf{T}_{th}^{-1} \\ &= \mathbf{T}_{th} \exp(-(\xi_{hw}^\wedge + \delta\xi_{hw}^\wedge)) \mathbf{T}_{th}^{-1} \\ &= \exp(-\text{Ad}_{\mathbf{T}_{th}}(\xi_{hw}^\wedge + \delta\xi_{hw}^\wedge)) \\ \therefore \exp(\xi_{th}^\wedge + \delta\xi_{th}^\wedge) &= \exp(-\text{Ad}_{\mathbf{T}_{th}}(\xi_{hw}^\wedge + \delta\xi_{hw}^\wedge)) \\ \therefore \frac{\partial \xi_{th}}{\partial \xi_{hw}} &= -\text{Ad}_{\mathbf{T}_{th}} \end{aligned}$$

$$\delta\xi^\wedge = \begin{bmatrix} \delta\mathbf{w}^\wedge & \delta\mathbf{v} \\ \mathbf{0}^T & 0 \end{bmatrix} \in \mathbb{R}^{4 \times 4}$$

$$\delta\xi = \begin{bmatrix} \delta\mathbf{v} \\ \delta\mathbf{w} \end{bmatrix} = \begin{bmatrix} \delta v_x \\ \delta v_y \\ \delta v_z \\ \delta w_x \\ \delta w_y \\ \delta w_z \end{bmatrix} \in \mathbb{R}^6$$

$\delta\xi \in \mathbb{R}^6$	상대 포즈 변화량 (twist)(6차원벡터)
$\delta\xi^\wedge \in \text{se}(3)$	hat 연산자가 적용된 twist (4x4행렬)(lie algebra)
$\exp(\xi^\wedge) \in \text{SE}(3)$	3차원 포즈 (transform)(4x4행렬)(lie group)

Lie Theory for Robotics 관련 내용은 다음 자료 참조: <http://ethaneade.com/lie.pdf>

more detail: <https://www.cnblogs.com/JingeTU/p/8306727.html>

4.3.4. Keyframes → Sliding Window Optimization → Adjoint Transformation

$\text{Ad}_{\mathbf{T}_{th}}$ 의 자세한 정의 및 역할에 대해 자세히 알고 싶으면 우측 하단의 링크를 참조하면 된다.

$$\frac{\partial \xi_{th}}{\partial \xi_{hw}} = -\text{Ad}_{\mathbf{T}_{th}}$$

$$\frac{\partial \xi_{th}}{\partial \xi_{tw}} = \mathbf{I}$$

$$\exp(\text{Ad}_{\mathbf{T}_{th}} \cdot \xi^\wedge) = \mathbf{T}_{th} \exp(\xi^\wedge) \mathbf{T}_{th}^{-1}$$

해당 슬라이드에서는 $\frac{\partial \xi_{th}}{\partial \xi_{hw}}$ $\frac{\partial \xi_{th}}{\partial \xi_{tw}}$ 두 값의 유도 과정에 대해 설명한다.

우선 $\mathbf{T}_{th} = \mathbf{T}_{tw} \mathbf{T}_{hw}^{-1}$ 과 같이 2개의 포즈로 분할할 수 있고 이를 Iterative 업데이트 공식으로 확장하면 다음과 같다.

host

$$\begin{aligned} \exp(\xi_{th}^\wedge + \delta\xi_{th}^\wedge) \mathbf{T}_{th} &= \mathbf{T}_{tw} (\exp(\xi_{hw}^\wedge + \delta\xi_{hw}^\wedge) \mathbf{T}_{hw})^{-1} \\ \exp(\xi_{th}^\wedge + \delta\xi_{th}^\wedge) &= \mathbf{T}_{tw} \mathbf{T}_{hw}^{-1} \exp(-(\xi_{hw}^\wedge + \delta\xi_{hw}^\wedge)) \mathbf{T}_{th}^{-1} \\ &= \mathbf{T}_{th} \exp(-(\xi_{hw}^\wedge + \delta\xi_{hw}^\wedge)) \mathbf{T}_{th}^{-1} \\ &= \exp(-\text{Ad}_{\mathbf{T}_{th}}(\xi_{hw}^\wedge + \delta\xi_{hw}^\wedge)) \\ \therefore \exp(\xi_{th}^\wedge + \delta\xi_{th}^\wedge) &= \exp(-\text{Ad}_{\mathbf{T}_{th}}(\xi_{hw}^\wedge + \delta\xi_{hw}^\wedge)) \\ \therefore \frac{\partial \xi_{th}}{\partial \xi_{hw}} &= -\text{Ad}_{\mathbf{T}_{th}} \end{aligned}$$

target

$$\begin{aligned} \exp(\xi_{th}^\wedge + \delta\xi_{th}^\wedge) \mathbf{T}_{th} &= \exp(\xi_{tw}^\wedge + \delta\xi_{tw}^\wedge) \mathbf{T}_{tw} \mathbf{T}_{hw}^{-1} \\ \exp(\xi_{th}^\wedge + \delta\xi_{th}^\wedge) &= \exp(\xi_{tw}^\wedge + \delta\xi_{tw}^\wedge) \mathbf{T}_{tw} \mathbf{T}_{hw}^{-1} \mathbf{T}_{th}^{-1} \\ &= \exp(\xi_{tw}^\wedge + \delta\xi_{tw}^\wedge) \end{aligned}$$

$$\therefore \exp(\xi_{th}^\wedge + \delta\xi_{th}^\wedge) = \exp(\xi_{tw}^\wedge + \delta\xi_{tw}^\wedge)$$

$$\therefore \frac{\partial \xi_{th}}{\partial \xi_{tw}} = \mathbf{I}$$

$$\delta\xi^\wedge = \begin{bmatrix} \delta\mathbf{w}^\wedge & \delta\mathbf{v} \\ \mathbf{0}^T & 0 \end{bmatrix} \in \mathbb{R}^{4 \times 4}$$

$$\delta\xi = \begin{bmatrix} \delta\mathbf{v} \\ \delta\mathbf{w} \end{bmatrix} = \begin{bmatrix} \delta v_x \\ \delta v_y \\ \delta v_z \\ \delta w_x \\ \delta w_y \\ \delta w_z \end{bmatrix} \in \mathbb{R}^6$$

$$\delta\xi \in \mathbb{R}^6$$

상대 포즈 변화량
(twist)(6차원벡터)

$$\delta\xi^\wedge \in \text{se}(3)$$

hat 연산자가 적용된 twist
(4x4행렬)(lie algebra)

$$\exp(\xi^\wedge) \in \text{SE}(3)$$

3차원 포즈
(transform)(4x4행렬)(lie group)

Lie Theory for Robotics 관련 내용은 다음 자료 참조: <http://ethaneade.com/lie.pdf>

more detail: <https://www.cnblogs.com/JingeTU/p/8306727.html>

4.3.5. Keyframes → Sliding Window Optimization → Marginalization

To Be Added

4.3.6. Keyframes → Sliding Window Optimization → Solving The Incremental Equation

DSO에서는 Sliding Window 내부의 항상 8개의 키프레임을 유지하면서

Pose(6) + Photometric(2) + Camera Intrinsics(4) + Idepth(N)의 총 $12+N$ 개의 파라미터를 최적화한다.

4.3.6. Keyframes → Sliding Window Optimization → Solving The Incremental Equation

DSO에서는 Sliding Window 내부의 항상 8개의 키프레임을 유지하면서

Pose(6) + Photometric(2) + Camera Intrinsics(4) + Idepth(N)의 총 $12+N$ 개의 파라미터를 최적화한다.

이렇게 특정 영역 내의 키프레임의 포즈와 맵포인트를 동시에 최적화하는 방법을 Local Bundle Adjustment(LBA)라고 한다.

설명의 편의상 이후 섹션에서는 Sliding Window Optimization을 LBA라고 언급한다.

4.3.6. Keyframes → Sliding Window Optimization → Solving The Incremental Equation

DSO에서는 Sliding Window 내부의 항상 8개의 키프레임을 유지하면서

Pose(6) + Photometric(2) + Camera Intrinsics(4) + Idepth(N)의 총 12+N개의 파라미터를 최적화한다.

이렇게 특정 영역 내의 키프레임의 포즈와 맵포인트를 동시에 최적화하는 방법을 Local Bundle Adjustment(LBA)라고 한다.

설명의 편의상 이후 섹션에서는 Sliding Window Optimization을 LBA라고 언급한다.

이전 섹션에서 유도한 LBA의 에러함수는 다음과 같고 이 때 state variable \mathbf{x} 는 다음과 같다.

$$f(\mathbf{x}) = \sqrt{w_h} \mathbf{r}$$

$$\mathbf{x} = \begin{bmatrix} \rho_1^{(1)} & \dots & \rho_1^{(N)} & \delta\xi^T & a & b & \mathbf{c} \end{bmatrix}_{(N+12) \times 1}^T$$

$$= \begin{bmatrix} \rho_1^{(1)} & \dots & \rho_1^{(N)} & \mathbf{y} \end{bmatrix}_{(N+12) \times 1}^T$$

$\rho_1^{(i)}$	1번 이미지에서 i번째 inverse depth 값 (N개)
$\delta\xi$	두 카메라 간 상대 포즈 변화량 (twist)(6차원 벡터) (6개)
a, b	두 이미지 간 exposure time을 고려하여 이미지 밝기를 조절하는 파라미터 (2개)
\mathbf{c}	카메라 내부 파라미터 (f_x, f_y, c_x, c_y) (4개)

4.3.6. Keyframes → Sliding Window Optimization → Solving The Incremental Equation

DSO에서는 Sliding Window 내부의 항상 8개의 키프레임을 유지하면서

Pose(6) + Photometric(2) + Camera Intrinsics(4) + Idepth(N)의 총 12+N개의 파라미터를 최적화한다.

이렇게 특정 영역 내의 키프레임의 포즈와 맵포인트를 동시에 최적화하는 방법을 Local Bundle Adjustment(LBA)라고 한다.

설명의 편의상 이후 섹션에서는 Sliding Window Optimization을 LBA라고 언급한다.

이전 섹션에서 유도한 LBA의 에러함수는 다음과 같고 이 때 state variable \mathbf{x} 는 다음과 같다.

$$f(\mathbf{x}) = \sqrt{w_h} \mathbf{r}$$

$$\mathbf{x} = \begin{bmatrix} \rho_1^{(1)} & \dots & \rho_1^{(N)} & \delta\xi^T & a & b & \mathbf{c} \end{bmatrix}_{(N+12) \times 1}^T$$

$$= \begin{bmatrix} \rho_1^{(1)} & \dots & \rho_1^{(N)} & \mathbf{y} \end{bmatrix}_{(N+12) \times 1}^T$$

$\rho_1^{(i)}$	1번 이미지에서 i번째 inverse depth 값 (N개)
$\delta\xi$	두 카메라 간 상대 포즈 변화량 (twist)(6차원 벡터) (6개)
a, b	두 이미지 간 exposure time을 고려하여 이미지 밝기를 조절하는 파라미터 (2개)
\mathbf{c}	카메라 내부 파라미터 (f_x, f_y, c_x, c_y) (4개)

수식을 간결하게 나타내기 위해 Idepth를 제외한 나머지 파라미터를 $\mathbf{y} = [\delta\xi^T \ a \ b \ \mathbf{c}]$ 로 축약하였다.

4.3.6. Keyframes → Sliding Window Optimization → Solving The Incremental Equation

해당 에러함수를 이용해 Least Square Optimization 형태로 나타내면 최종적으로 다음과 같은 형태의 식이 유도된다.

해당 식의 유도는 1장에서 자세히 다뤘기 때문에 생략한다.

$$\boxed{\mathbf{H}\Delta\mathbf{x} = -\mathbf{b}} \quad \rightarrow \quad \boxed{\begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho\mathbf{y}} \\ \mathbf{H}_{\mathbf{y}\rho} & \mathbf{H}_{\mathbf{y}\mathbf{y}} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{x}_\rho \\ \Delta\mathbf{x}_\mathbf{y} \end{bmatrix} = \begin{bmatrix} -\mathbf{b}_\rho \\ -\mathbf{b}_\mathbf{y} \end{bmatrix}} \quad \text{where, } \mathbf{y} = [\delta\xi^T \ a \ b \ \mathbf{c}]$$

4.3.6. Keyframes → Sliding Window Optimization → Solving The Incremental Equation

해당 에러함수를 이용해 Least Square Optimization 형태로 나타내면 최종적으로 다음과 같은 형태의 식이 유도된다.

해당 식의 유도는 1장에서 자세히 다뤘기 때문에 생략한다.

$$\mathbf{H}\Delta\mathbf{x} = -\mathbf{b} \rightarrow \begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho y} \\ \mathbf{H}_{y\rho} & \mathbf{H}_{yy} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{x}_\rho \\ \Delta\mathbf{x}_y \end{bmatrix} = \begin{bmatrix} -\mathbf{b}_\rho \\ -\mathbf{b}_y \end{bmatrix} \text{ where, } \mathbf{y} = [\delta\xi^T \ a \ b \ \mathbf{c}]$$

이 때 Hessian Matrix를 자세하게 펼쳐보면 다음과 같다.

$$\mathbf{H} = \mathbf{J}^T \mathbf{J} = \begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho y} \\ \mathbf{H}_{y\rho} & \mathbf{H}_{yy} \end{bmatrix} = \begin{bmatrix} \text{idepth (N)} & & \\ & \text{idepth (N)} & \\ & & 1 \times 1 \\ \text{idepth (N)} & & \\ & & & \text{pose + photometric + intrinsics (12)} \\ & & & & \text{pose + photometric + intrinsics (12)} \\ & & & & & 12 \times 12 \end{bmatrix}$$

4.3.6. Keyframes → Sliding Window Optimization → Solving The Incremental Equation

해당 에러함수를 이용해 Least Square Optimization 형태로 나타내면 최종적으로 다음과 같은 형태의 식이 유도된다.

해당 식의 유도는 1장에서 자세히 다뤘기 때문에 생략한다.

$$\mathbf{H}\Delta\mathbf{x} = -\mathbf{b} \rightarrow \begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho y} \\ \mathbf{H}_{y\rho} & \mathbf{H}_{yy} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{x}_\rho \\ \Delta\mathbf{x}_y \end{bmatrix} = \begin{bmatrix} -\mathbf{b}_\rho \\ -\mathbf{b}_y \end{bmatrix} \text{ where, } \mathbf{y} = [\delta\xi^T \ a \ b \ \mathbf{c}]$$

이 때 Hessian Matrix를 자세하게 펼쳐보면 다음과 같다.

$$\mathbf{H} = \mathbf{J}^T \mathbf{J} = \begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho y} \\ \mathbf{H}_{y\rho} & \mathbf{H}_{yy} \end{bmatrix} = \begin{bmatrix} \text{idepth (N)} & & \\ & \text{idepth (N)} & \\ & & 1 \times 1 \\ \text{idepth (N)} & & \\ & & & \text{pose + photometric + intrinsics (12)} \\ & & & & \text{pose + photometric + intrinsics (12)} \\ & & & & & 12 \times 12 \end{bmatrix}$$

이 때, Initialization 섹션에서 언급한 것처럼 Inverse Depth(**N개**)의 경우 최소 수 백 ~ 수 천개의 맵포인트에 대한 정보를 포함하고 있으므로 해당 Hessian Matrix의 역행렬을 계산하는데 매우 많은 시간이 소모된다.

따라서 이를 효율적으로 계산하기 위해 Schur Complement를 사용해야 한다.

4.3.6. Keyframes → Sliding Window Optimization → Solving The Incremental Equation

Schur Complement는 Initialization 섹션에서 이미 설명했으나 중요한 내용이므로 LBA 과정에서도 반복해서 설명한다.

$$\begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho\mathbf{y}} \\ \mathbf{H}_{\mathbf{y}\rho} & \mathbf{H}_{\mathbf{y}\mathbf{y}} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x}_\rho \\ \Delta \mathbf{x}_\mathbf{y} \end{bmatrix} = \begin{bmatrix} \mathbf{b}_\rho \\ \mathbf{b}_\mathbf{y} \end{bmatrix}$$

$$\mathbf{y} = [\delta\xi^T \ a \ b \ \mathbf{c}]$$

$$\mathbf{H}_{\rho\rho} = \sum \mathbf{J}_\rho^T \mathbf{J}_\rho$$

$$\mathbf{H}_{\rho\mathbf{y}} = \sum \mathbf{J}_\rho^T \mathbf{J}_\mathbf{y}$$

$$\mathbf{H}_{\mathbf{y}\rho} = \sum \mathbf{J}_\mathbf{y}^T \mathbf{J}_\rho$$

$$\mathbf{H}_{\mathbf{y}\mathbf{y}} = \sum \mathbf{J}_\mathbf{y}^T \mathbf{J}_\mathbf{y}$$

$$\mathbf{b}_\rho = - \sum \mathbf{J}_\rho^T f(\mathbf{x})$$

$$\mathbf{b}_\mathbf{y} = - \sum \mathbf{J}_\mathbf{y}^T f(\mathbf{x})$$

$$\mathbf{H}_{sc} = \mathbf{H}_{\mathbf{y}\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{H}_{\rho\mathbf{y}}$$

$$\mathbf{b}_{sc} = \mathbf{H}_{\mathbf{y}\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{b}_\rho$$

4.3.6. Keyframes → Sliding Window Optimization → Solving The Incremental Equation

Schur Complement는 Initialization 섹션에서 이미 설명했으나 중요한 내용이므로 LBA 과정에서도 반복해서 설명한다.

$$\begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho y} \\ \mathbf{H}_{y\rho} & \mathbf{H}_{yy} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x}_\rho \\ \Delta \mathbf{x}_y \end{bmatrix} = \begin{bmatrix} \mathbf{b}_\rho \\ \mathbf{b}_y \end{bmatrix}$$

위 식에 양변에 아래와 같은 특수한 형태의 행렬을 양변에 곱해준다.



$$\begin{bmatrix} \mathbf{I} & \mathbf{0} \\ -\mathbf{H}_{y\rho}\mathbf{H}_{\rho\rho}^{-1} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho y} \\ \mathbf{H}_{y\rho} & \mathbf{H}_{yy} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x}_\rho \\ \Delta \mathbf{x}_y \end{bmatrix} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ -\mathbf{H}_{y\rho}\mathbf{H}_{\rho\rho}^{-1} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{b}_\rho \\ \mathbf{b}_y \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho y} \\ \mathbf{0} & -\mathbf{H}_{y\rho}\mathbf{H}_{\rho\rho}^{-1}\mathbf{H}_{\rho y} + \mathbf{H}_{yy} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x}_\rho \\ \Delta \mathbf{x}_y \end{bmatrix} = \begin{bmatrix} \mathbf{b}_\rho \\ -\mathbf{H}_{y\rho}\mathbf{H}_{\rho\rho}^{-1}\mathbf{b}_\rho + \mathbf{b}_y \end{bmatrix}$$

$\mathbf{y} = [\delta\xi^T \ a \ b \ \mathbf{c}]$
$\mathbf{H}_{\rho\rho} = \sum \mathbf{J}_\rho^T \mathbf{J}_\rho$
$\mathbf{H}_{\rho y} = \sum \mathbf{J}_\rho^T \mathbf{J}_y$
$\mathbf{H}_{y\rho} = \sum \mathbf{J}_y^T \mathbf{J}_\rho$
$\mathbf{H}_{yy} = \sum \mathbf{J}_y^T \mathbf{J}_y$
$\mathbf{b}_\rho = -\sum \mathbf{J}_\rho^T f(\mathbf{x})$
$\mathbf{b}_y = -\sum \mathbf{J}_y^T f(\mathbf{x})$
$\mathbf{H}_{sc} = \mathbf{H}_{y\rho}\mathbf{H}_{\rho\rho}^{-1}\mathbf{H}_{\rho y}$
$\mathbf{b}_{sc} = \mathbf{H}_{y\rho}\mathbf{H}_{\rho\rho}^{-1}\mathbf{b}_\rho$

4.3.6. Keyframes → Sliding Window Optimization → Solving The Incremental Equation

Schur Complement는 Initialization 섹션에서 이미 설명했으나 중요한 내용이므로 LBA 과정에서도 반복해서 설명한다.

$$\begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho y} \\ \mathbf{H}_{y\rho} & \mathbf{H}_{yy} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x}_\rho \\ \Delta \mathbf{x}_y \end{bmatrix} = \begin{bmatrix} \mathbf{b}_\rho \\ \mathbf{b}_y \end{bmatrix}$$

위 식에 양변에 아래와 같은 특수한 형태의 행렬을 양변에 곱해준다.

$$\begin{aligned} \mathbf{y} &= [\delta\xi^T \ a \ b \ \mathbf{c}] \\ \mathbf{H}_{\rho\rho} &= \sum \mathbf{J}_\rho^T \mathbf{J}_\rho \\ \mathbf{H}_{\rho y} &= \sum \mathbf{J}_\rho^T \mathbf{J}_y \\ \mathbf{H}_{y\rho} &= \sum \mathbf{J}_y^T \mathbf{J}_\rho \\ \mathbf{H}_{yy} &= \sum \mathbf{J}_y^T \mathbf{J}_y \\ \mathbf{b}_\rho &= -\sum \mathbf{J}_\rho^T f(\mathbf{x}) \\ \mathbf{b}_y &= -\sum \mathbf{J}_y^T f(\mathbf{x}) \\ \mathbf{H}_{sc} &= \mathbf{H}_{y\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{H}_{\rho y} \\ \mathbf{b}_{sc} &= \mathbf{H}_{y\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{b}_\rho \end{aligned}$$

$$\begin{bmatrix} \mathbf{I} & \mathbf{0} \\ -\mathbf{H}_{y\rho} \mathbf{H}_{\rho\rho}^{-1} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho y} \\ \mathbf{H}_{y\rho} & \mathbf{H}_{yy} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x}_\rho \\ \Delta \mathbf{x}_y \end{bmatrix} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ -\mathbf{H}_{y\rho} \mathbf{H}_{\rho\rho}^{-1} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{b}_\rho \\ \mathbf{b}_y \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho y} \\ \mathbf{0} & -\mathbf{H}_{y\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{H}_{\rho y} + \mathbf{H}_{yy} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x}_\rho \\ \Delta \mathbf{x}_y \end{bmatrix} = \begin{bmatrix} \mathbf{b}_\rho \\ -\mathbf{H}_{y\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{b}_\rho + \mathbf{b}_y \end{bmatrix}$$

위 식을 전개해서 $\Delta \mathbf{x}_y$ $\Delta \mathbf{x}_\rho$ 의 값을 순차적으로 계산할 수 있다. 위 식을 전개하면 $\Delta \mathbf{x}_y$ 만 존재하는 항이 유도된다.

1 $\Delta \mathbf{x}_y = (\mathbf{H}_{yy} - \mathbf{H}_{sc})^{-1} (\mathbf{b}_y - \mathbf{b}_{sc})$

4.3.6. Keyframes → Sliding Window Optimization → Solving The Incremental Equation

Schur Complement는 Initialization 섹션에서 이미 설명했으나 중요한 내용이므로 LBA 과정에서도 반복해서 설명한다.

$$\begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho y} \\ \mathbf{H}_{y\rho} & \mathbf{H}_{yy} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x}_\rho \\ \Delta \mathbf{x}_y \end{bmatrix} = \begin{bmatrix} \mathbf{b}_\rho \\ \mathbf{b}_y \end{bmatrix}$$

위 식에 양변에 아래와 같은 특수한 형태의 행렬을 양변에 곱해준다.

$$\begin{bmatrix} \mathbf{I} & \mathbf{0} \\ -\mathbf{H}_{y\rho}\mathbf{H}_{\rho\rho}^{-1} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho y} \\ \mathbf{H}_{y\rho} & \mathbf{H}_{yy} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x}_\rho \\ \Delta \mathbf{x}_y \end{bmatrix} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ -\mathbf{H}_{y\rho}\mathbf{H}_{\rho\rho}^{-1} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{b}_\rho \\ \mathbf{b}_y \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho y} \\ \mathbf{0} & -\mathbf{H}_{y\rho}\mathbf{H}_{\rho\rho}^{-1}\mathbf{H}_{\rho y} + \mathbf{H}_{yy} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x}_\rho \\ \Delta \mathbf{x}_y \end{bmatrix} = \begin{bmatrix} \mathbf{b}_\rho \\ -\mathbf{H}_{y\rho}\mathbf{H}_{\rho\rho}^{-1}\mathbf{b}_\rho + \mathbf{b}_y \end{bmatrix}$$

위 식을 전개해서 $\Delta \mathbf{x}_y$ $\Delta \mathbf{x}_\rho$ 의 값을 순차적으로 계산할 수 있다. 위 식을 전개하면 $\Delta \mathbf{x}_y$ 만 존재하는 항이 유도된다.

$$① \quad \Delta \mathbf{x}_y = (\mathbf{H}_{yy} - \mathbf{H}_{sc})^{-1}(\mathbf{b}_y - \mathbf{b}_{sc})$$

위 식을 통해 $\Delta \mathbf{x}_y$ 를 먼저 계산한 후 이를 토대로 $\Delta \mathbf{x}_\rho$ 를 계산한다. 이를 통해 역행렬을 구하는 것보다 더욱 빠르게 최적화 공식을 풀 수 있다.

$$② \quad \Delta \mathbf{x}_\rho = \mathbf{H}_{\rho\rho}^{-1}(\mathbf{b}_\rho - \mathbf{H}_{\rho y}\Delta \mathbf{x}_y)$$

$\mathbf{y} = [\delta\xi^T \ a \ b \ \mathbf{c}]$
$\mathbf{H}_{\rho\rho} = \sum \mathbf{J}_\rho^T \mathbf{J}_\rho$
$\mathbf{H}_{\rho y} = \sum \mathbf{J}_\rho^T \mathbf{J}_y$
$\mathbf{H}_{y\rho} = \sum \mathbf{J}_y^T \mathbf{J}_\rho$
$\mathbf{H}_{yy} = \sum \mathbf{J}_y^T \mathbf{J}_y$
$\mathbf{b}_\rho = -\sum \mathbf{J}_\rho^T f(\mathbf{x})$
$\mathbf{b}_y = -\sum \mathbf{J}_y^T f(\mathbf{x})$
$\mathbf{H}_{sc} = \mathbf{H}_{y\rho}\mathbf{H}_{\rho\rho}^{-1}\mathbf{H}_{\rho y}$
$\mathbf{b}_{sc} = \mathbf{H}_{y\rho}\mathbf{H}_{\rho\rho}^{-1}\mathbf{b}_\rho$

4.3.6. Keyframes → Sliding Window Optimization → Solving The Incremental Equation

Inverse depth와 관련 있는 항인 $\mathbf{H}_{\rho\rho}$ 는 $N \times N$ 크기의 대각행렬(diagonal matrix)이므로 쉽게 역행렬을 구할 수 있다.

따라서 $\mathbf{H}_{sc}, \mathbf{b}_{sc}$ 또한 다음과 같이 비교적 간단하게 계산할 수 있다.

$$\mathbf{H}_{sc} = \mathbf{H}_{y\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{H}_{\rho y} = \sum \frac{1}{\mathbf{J}_\rho \mathbf{J}_\rho^T} (\mathbf{J}_\rho^T \mathbf{J}_y)^T \mathbf{J}_\rho^T \mathbf{J}_y$$

$$\mathbf{b}_{sc} = \mathbf{H}_{y\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{b}_\rho = - \sum \frac{1}{\mathbf{J}_\rho \mathbf{J}_\rho^T} (\mathbf{J}_\rho^T \mathbf{J}_y)^T \mathbf{J}_\rho^T f(\mathbf{x})$$

$$\mathbf{y} = [\delta\xi^T \ a \ b \ \mathbf{c}]$$

$$\mathbf{H}_{\rho\rho} = \sum \mathbf{J}_\rho^T \mathbf{J}_\rho$$

$$\mathbf{H}_{\rho y} = \sum \mathbf{J}_\rho^T \mathbf{J}_y$$

$$\mathbf{H}_{y\rho} = \sum \mathbf{J}_y^T \mathbf{J}_\rho$$

$$\mathbf{H}_{yy} = \sum \mathbf{J}_y^T \mathbf{J}_y$$

$$\mathbf{b}_\rho = - \sum \mathbf{J}_\rho^T f(\mathbf{x})$$

$$\mathbf{b}_y = - \sum \mathbf{J}_y^T f(\mathbf{x})$$

$$\mathbf{H}_{sc} = \mathbf{H}_{y\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{H}_{\rho y}$$

$$\mathbf{b}_{sc} = \mathbf{H}_{y\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{b}_\rho$$

4.3.6. Keyframes → Sliding Window Optimization → Solving The Incremental Equation

Inverse depth와 관련 있는 항인 $\mathbf{H}_{\rho\rho}$ 는 $N \times N$ 크기의 대각행렬(diagonal matrix)이므로 쉽게 역행렬을 구할 수 있다.

따라서 $\mathbf{H}_{sc}, \mathbf{b}_{sc}$ 또한 다음과 같이 비교적 간단하게 계산할 수 있다.

$$\mathbf{H}_{sc} = \mathbf{H}_{y\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{H}_{\rho y} = \sum \frac{1}{\mathbf{J}_\rho \mathbf{J}_\rho^T} (\mathbf{J}_\rho^T \mathbf{J}_y)^T \mathbf{J}_\rho^T \mathbf{J}_y$$

$$\mathbf{b}_{sc} = \mathbf{H}_{y\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{b}_\rho = - \sum \frac{1}{\mathbf{J}_\rho \mathbf{J}_\rho^T} (\mathbf{J}_\rho^T \mathbf{J}_y)^T \mathbf{J}_\rho^T f(\mathbf{x})$$

$$\begin{aligned}\mathbf{y} &= [\delta\xi^T \ a \ b \ \mathbf{c}] \\ \mathbf{H}_{\rho\rho} &= \sum \mathbf{J}_\rho^T \mathbf{J}_\rho \\ \mathbf{H}_{\rho y} &= \sum \mathbf{J}_\rho^T \mathbf{J}_y \\ \mathbf{H}_{y\rho} &= \sum \mathbf{J}_y^T \mathbf{J}_\rho \\ \mathbf{H}_{yy} &= \sum \mathbf{J}_y^T \mathbf{J}_y \\ \mathbf{b}_\rho &= - \sum \mathbf{J}_\rho^T f(\mathbf{x}) \\ \mathbf{b}_y &= - \sum \mathbf{J}_y^T f(\mathbf{x}) \\ \mathbf{H}_{sc} &= \mathbf{H}_{y\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{H}_{\rho y} \\ \mathbf{b}_{sc} &= \mathbf{H}_{y\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{b}_\rho\end{aligned}$$

최종적으로 damping 계수 λ 기반의 Levenberg-Marquardt 방법을 사용하여 Optimization을 수행한다.

$$(\mathbf{H} + \lambda \mathbf{I}) \Delta \mathbf{x} = \mathbf{b}$$

4.3.6. Keyframes → Sliding Window Optimization → Solving The Incremental Equation

$$\mathbf{y} = [\delta\xi^T \ a \ b \ \mathbf{c}]$$

\mathbf{H} , \mathbf{b} 행렬에 대해 자세하게 표현하면 다음과 같다.

$$\begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho\mathbf{y}} \\ \mathbf{H}_{\mathbf{y}\rho} & \mathbf{H}_{\mathbf{y}\mathbf{y}} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{x}_\rho \\ \Delta\mathbf{x}_\mathbf{y} \end{bmatrix} = \begin{bmatrix} -\mathbf{b}_\rho \\ -\mathbf{b}_\mathbf{y} \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho\mathbf{y}} \\ \mathbf{H}_{\mathbf{y}\rho} & \mathbf{H}_{\mathbf{y}\mathbf{y}} \end{bmatrix} = \begin{bmatrix} \mathbf{J}_\rho^T \mathbf{J}_\rho & \mathbf{J}_\rho^T \mathbf{J}_\mathbf{y} \\ \mathbf{J}_\mathbf{y}^T \mathbf{J}_\rho & \mathbf{J}_\mathbf{y}^T \mathbf{J}_\mathbf{y} \end{bmatrix}$$

$$\mathbf{H}_{\mathbf{y}\mathbf{y}} = \mathbf{J}_\mathbf{y}^T \mathbf{J}_\mathbf{y} = \begin{bmatrix} \mathbf{J}_\mathbf{c}^T \mathbf{J}_\mathbf{c} & \mathbf{J}_\mathbf{c}^T \mathbf{J}_{\xi'} \\ \mathbf{J}_{\xi'}^T \mathbf{J}_\mathbf{c} & \mathbf{J}_{\xi'}^T \mathbf{J}_{\xi'} \end{bmatrix} \quad \text{where, } \xi' = [\delta\xi \ a \ b]_{8 \times 1}$$

pose(6) + photo(2)

more detail: <https://www.cnblogs.com/JingeTU/p/8395046.html>

4.3.6. Keyframes → Sliding Window Optimization → Solving The Incremental Equation

$$\mathbf{y} = [\delta\xi^T \ a \ b \ \mathbf{c}]$$

\mathbf{H} , \mathbf{b} 행렬에 대해 자세하게 표현하면 다음과 같다.

$$\begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho\mathbf{y}} \\ \mathbf{H}_{\mathbf{y}\rho} & \mathbf{H}_{\mathbf{y}\mathbf{y}} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{x}_\rho \\ \Delta\mathbf{x}_\mathbf{y} \end{bmatrix} = \begin{bmatrix} -\mathbf{b}_\rho \\ -\mathbf{b}_\mathbf{y} \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho\mathbf{y}} \\ \mathbf{H}_{\mathbf{y}\rho} & \mathbf{H}_{\mathbf{y}\mathbf{y}} \end{bmatrix} = \begin{bmatrix} \mathbf{J}_\rho^T \mathbf{J}_\rho & \mathbf{J}_\rho^T \mathbf{J}_\mathbf{y} \\ \mathbf{J}_\mathbf{y}^T \mathbf{J}_\rho & \mathbf{J}_\mathbf{y}^T \mathbf{J}_\mathbf{y} \end{bmatrix}$$

$$\mathbf{H}_{\mathbf{y}\mathbf{y}} = \mathbf{J}_\mathbf{y}^T \mathbf{J}_\mathbf{y} = \begin{bmatrix} \mathbf{J}_\mathbf{c}^T \mathbf{J}_\mathbf{c} & \mathbf{J}_\mathbf{c}^T \mathbf{J}_{\xi'} \\ \mathbf{J}_{\xi'}^T \mathbf{J}_\mathbf{c} & \mathbf{J}_{\xi'}^T \mathbf{J}_{\xi'} \end{bmatrix} \quad \text{where, } \xi' = [\delta\xi \ a \ b]_{8 \times 1}$$

$$\mathbf{J}_\mathbf{c}^T \mathbf{J}_\mathbf{c} = \begin{bmatrix} \frac{\partial \mathbf{r}^{(1)}}{\partial \mathbf{c}}^T & \cdots & \frac{\partial \mathbf{r}^{(N)}}{\partial \mathbf{c}}^T \end{bmatrix} \begin{bmatrix} \frac{\partial \mathbf{r}^{(1)}}{\partial \mathbf{c}} \\ \vdots \\ \frac{\partial \mathbf{r}^{(N)}}{\partial \mathbf{c}} \end{bmatrix} = \sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \mathbf{c}}^T \frac{\partial \mathbf{r}^{(i)}}{\partial \mathbf{c}}$$

more detail: <https://www.cnblogs.com/JingeTU/p/8395046.html>

4.3.6. Keyframes → Sliding Window Optimization → Solving The Incremental Equation

$$\mathbf{y} = [\delta\xi^T \ a \ b \ \mathbf{c}]$$

\mathbf{H} , \mathbf{b} 행렬에 대해 자세하게 표현하면 다음과 같다.

$$\begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho\mathbf{y}} \\ \mathbf{H}_{\mathbf{y}\rho} & \mathbf{H}_{\mathbf{y}\mathbf{y}} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{x}_\rho \\ \Delta\mathbf{x}_\mathbf{y} \end{bmatrix} = \begin{bmatrix} -\mathbf{b}_\rho \\ -\mathbf{b}_\mathbf{y} \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho\mathbf{y}} \\ \mathbf{H}_{\mathbf{y}\rho} & \mathbf{H}_{\mathbf{y}\mathbf{y}} \end{bmatrix} = \begin{bmatrix} \mathbf{J}_\rho^T \mathbf{J}_\rho & \mathbf{J}_\rho^T \mathbf{J}_\mathbf{y} \\ \mathbf{J}_\mathbf{y}^T \mathbf{J}_\rho & \mathbf{J}_\mathbf{y}^T \mathbf{J}_\mathbf{y} \end{bmatrix}$$

$$\mathbf{H}_{\mathbf{y}\mathbf{y}} = \mathbf{J}_\mathbf{y}^T \mathbf{J}_\mathbf{y} = \begin{bmatrix} \mathbf{J}_\mathbf{c}^T \mathbf{J}_\mathbf{c} & \mathbf{J}_\mathbf{c}^T \mathbf{J}_{\xi'} \\ \mathbf{J}_{\xi'}^T \mathbf{J}_\mathbf{c} & \mathbf{J}_{\xi'}^T \mathbf{J}_{\xi'} \end{bmatrix} \quad \text{where, } \xi' = [\delta\xi \ a \ b]_{8 \times 1}$$

$$\mathbf{J}_\mathbf{c}^T \mathbf{J}_\mathbf{c} = \left[\frac{\partial \mathbf{r}^{(1)}}{\partial \mathbf{c}}^T \ \dots \ \frac{\partial \mathbf{r}^{(N)}}{\partial \mathbf{c}}^T \right] \begin{bmatrix} \frac{\partial \mathbf{r}^{(1)}}{\partial \mathbf{c}} \\ \vdots \\ \frac{\partial \mathbf{r}^{(N)}}{\partial \mathbf{c}} \end{bmatrix} = \sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \mathbf{c}}^T \frac{\partial \mathbf{r}^{(i)}}{\partial \mathbf{c}}$$

$$\mathbf{J}_\mathbf{c}^T \mathbf{J}_{\xi'} = \left[\frac{\partial \mathbf{r}^{(1)}}{\partial \mathbf{c}}^T \ \dots \ \frac{\partial \mathbf{r}^{(N)}}{\partial \mathbf{c}}^T \right] \begin{bmatrix} \frac{\partial \mathbf{r}^{(1)}}{\partial \xi_1} & \dots & \frac{\partial \mathbf{r}^{(1)}}{\partial \xi_8} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathbf{r}^{(N)}}{\partial \xi_1} & \dots & \frac{\partial \mathbf{r}^{(N)}}{\partial \xi_8} \end{bmatrix} = \left[\sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \mathbf{c}}^T \frac{\partial \mathbf{r}^{(i)}}{\partial \xi_1} \ \dots \ \sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \mathbf{c}}^T \frac{\partial \mathbf{r}^{(i)}}{\partial \xi_8} \right]$$

more detail: <https://www.cnblogs.com/JingeTU/p/8395046.html>

4.3.6. Keyframes → Sliding Window Optimization → Solving The Incremental Equation

$$\mathbf{y} = [\delta\xi^T \ a \ b \ \mathbf{c}]$$

\mathbf{H} , \mathbf{b} 행렬에 대해 자세하게 표현하면 다음과 같다.

$$\begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho\mathbf{y}} \\ \mathbf{H}_{\mathbf{y}\rho} & \mathbf{H}_{\mathbf{y}\mathbf{y}} \end{bmatrix} = \begin{bmatrix} \mathbf{J}_\rho^T \mathbf{J}_\rho & \mathbf{J}_\rho^T \mathbf{J}_\mathbf{y} \\ \mathbf{J}_\mathbf{y}^T \mathbf{J}_\rho & \mathbf{J}_\mathbf{y}^T \mathbf{J}_\mathbf{y} \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho\mathbf{y}} \\ \mathbf{H}_{\mathbf{y}\rho} & \mathbf{H}_{\mathbf{y}\mathbf{y}} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x}_\rho \\ \Delta \mathbf{x}_\mathbf{y} \end{bmatrix} = \begin{bmatrix} -\mathbf{b}_\rho \\ -\mathbf{b}_\mathbf{y} \end{bmatrix}$$

$$\mathbf{H}_{\mathbf{y}\mathbf{y}} = \mathbf{J}_\mathbf{y}^T \mathbf{J}_\mathbf{y} = \begin{bmatrix} \mathbf{J}_\mathbf{c}^T \mathbf{J}_\mathbf{c} & \mathbf{J}_\mathbf{c}^T \mathbf{J}_{\xi'} \\ \mathbf{J}_{\xi'}^T \mathbf{J}_\mathbf{c} & \mathbf{J}_{\xi'}^T \mathbf{J}_{\xi'} \end{bmatrix} \quad \text{where, } \xi' = [\delta\xi \ a \ b]_{8 \times 1}$$

$$\mathbf{J}_\mathbf{c}^T \mathbf{J}_\mathbf{c} = \left[\frac{\partial \mathbf{r}^{(1)}}{\partial \mathbf{c}}^T \ \dots \ \frac{\partial \mathbf{r}^{(N)}}{\partial \mathbf{c}}^T \right] \begin{bmatrix} \frac{\partial \mathbf{r}^{(1)}}{\partial \mathbf{c}} \\ \vdots \\ \frac{\partial \mathbf{r}^{(N)}}{\partial \mathbf{c}} \end{bmatrix} = \sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \mathbf{c}}^T \frac{\partial \mathbf{r}^{(i)}}{\partial \mathbf{c}}$$

$$\mathbf{J}_\mathbf{c}^T \mathbf{J}_{\xi'} = \left[\frac{\partial \mathbf{r}^{(1)}}{\partial \mathbf{c}}^T \ \dots \ \frac{\partial \mathbf{r}^{(N)}}{\partial \mathbf{c}}^T \right] \begin{bmatrix} \frac{\partial \mathbf{r}^{(1)}}{\partial \xi_1} & \dots & \frac{\partial \mathbf{r}^{(1)}}{\partial \xi_8} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathbf{r}^{(N)}}{\partial \xi_1} & \dots & \frac{\partial \mathbf{r}^{(N)}}{\partial \xi_8} \end{bmatrix} = \left[\sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \mathbf{c}}^T \frac{\partial \mathbf{r}^{(i)}}{\partial \xi_1} \ \dots \ \sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \mathbf{c}}^T \frac{\partial \mathbf{r}^{(i)}}{\partial \xi_8} \right]$$

$$\mathbf{J}_{\xi'}^T \mathbf{J}_{\xi'} = \begin{bmatrix} \frac{\partial \mathbf{r}^{(1)}}{\partial \xi_1} & \dots & \frac{\partial \mathbf{r}^{(1)}}{\partial \xi_8} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathbf{r}^{(N)}}{\partial \xi_1} & \dots & \frac{\partial \mathbf{r}^{(N)}}{\partial \xi_8} \end{bmatrix}^T \begin{bmatrix} \frac{\partial \mathbf{r}^{(1)}}{\partial \xi_1} & \dots & \frac{\partial \mathbf{r}^{(1)}}{\partial \xi_8} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathbf{r}^{(N)}}{\partial \xi_1} & \dots & \frac{\partial \mathbf{r}^{(N)}}{\partial \xi_8} \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \xi_1}^T \frac{\partial \mathbf{r}^{(i)}}{\partial \xi_1} & \dots & \sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \xi_1}^T \frac{\partial \mathbf{r}^{(i)}}{\partial \xi_8} \\ \vdots & \ddots & \vdots \\ \sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \xi_8}^T \frac{\partial \mathbf{r}^{(i)}}{\partial \xi_1} & \dots & \sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \xi_8}^T \frac{\partial \mathbf{r}^{(i)}}{\partial \xi_8} \end{bmatrix}$$

more detail: <https://www.cnblogs.com/JingeTU/p/8395046.html>

4.3.6. Keyframes → Sliding Window Optimization → Solving The Incremental Equation

$$\mathbf{y} = [\delta\xi^T \ a \ b \ \mathbf{c}]$$

\mathbf{H} , \mathbf{b} 행렬에 대해 자세하게 표현하면 다음과 같다.

$$\begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho\mathbf{y}} \\ \mathbf{H}_{\mathbf{y}\rho} & \mathbf{H}_{\mathbf{y}\mathbf{y}} \end{bmatrix} = \begin{bmatrix} \mathbf{J}_\rho^T \mathbf{J}_\rho & \mathbf{J}_\rho^T \mathbf{J}_\mathbf{y} \\ \mathbf{J}_\mathbf{y}^T \mathbf{J}_\rho & \mathbf{J}_\mathbf{y}^T \mathbf{J}_\mathbf{y} \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho\mathbf{y}} \\ \mathbf{H}_{\mathbf{y}\rho} & \mathbf{H}_{\mathbf{y}\mathbf{y}} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{x}_\rho \\ \Delta\mathbf{x}_\mathbf{y} \end{bmatrix} = \begin{bmatrix} -\mathbf{b}_\rho \\ -\mathbf{b}_\mathbf{y} \end{bmatrix}$$

more detail: <https://www.cnblogs.com/JingeTU/p/8395046.html>

4.3.6. Keyframes → Sliding Window Optimization → Solving The Incremental Equation

$$\mathbf{y} = [\delta\xi^T \ a \ b \ \mathbf{c}]$$

\mathbf{H} , \mathbf{b} 행렬에 대해 자세하게 표현하면 다음과 같다.

$$\begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho\mathbf{y}} \\ \mathbf{H}_{\mathbf{y}\rho} & \mathbf{H}_{\mathbf{y}\mathbf{y}} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{x}_\rho \\ \Delta\mathbf{x}_\mathbf{y} \end{bmatrix} = \begin{bmatrix} -\mathbf{b}_\rho \\ -\mathbf{b}_\mathbf{y} \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho\mathbf{y}} \\ \mathbf{H}_{\mathbf{y}\rho} & \mathbf{H}_{\mathbf{y}\mathbf{y}} \end{bmatrix} = \begin{bmatrix} \mathbf{J}_\rho^T \mathbf{J}_\rho & \mathbf{J}_\rho^T \mathbf{J}_\mathbf{y} \\ \mathbf{J}_\mathbf{y}^T \mathbf{J}_\rho & \mathbf{J}_\mathbf{y}^T \mathbf{J}_\mathbf{y} \end{bmatrix}$$

$$\mathbf{J}_\rho^T \mathbf{J}_\rho = \begin{bmatrix} \frac{\partial \mathbf{r}^{(1)}}{\partial \rho^{(1)}} & \cdots & \frac{\partial \mathbf{r}^{(1)}}{\partial \rho^{(M)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathbf{r}^{(N)}}{\partial \rho^{(1)}} & \cdots & \frac{\partial \mathbf{r}^{(N)}}{\partial \rho^{(M)}} \end{bmatrix}^T \begin{bmatrix} \frac{\partial \mathbf{r}^{(1)}}{\partial \rho^{(1)}} & \cdots & \frac{\partial \mathbf{r}^{(1)}}{\partial \rho^{(M)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathbf{r}^{(N)}}{\partial \rho^{(1)}} & \cdots & \frac{\partial \mathbf{r}^{(N)}}{\partial \rho^{(M)}} \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \rho^{(1)}}^T \frac{\partial \mathbf{r}^{(i)}}{\partial \rho^{(1)}} & 0 & \cdots & 0 \\ 0 & \sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \rho^{(2)}}^T \frac{\partial \mathbf{r}^{(i)}}{\partial \rho^{(2)}} & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \cdots & \sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \rho^{(M)}}^T \frac{\partial \mathbf{r}^{(i)}}{\partial \rho^{(M)}} \end{bmatrix}$$

4.3.6. Keyframes → Sliding Window Optimization → Solving The Incremental Equation

$$\mathbf{y} = [\delta\xi^T \ a \ b \ \mathbf{c}]$$

\mathbf{H} , \mathbf{b} 행렬에 대해 자세하게 표현하면 다음과 같다.

$$\begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho\mathbf{y}} \\ \mathbf{H}_{\mathbf{y}\rho} & \mathbf{H}_{\mathbf{y}\mathbf{y}} \end{bmatrix} = \begin{bmatrix} \mathbf{J}_\rho^T \mathbf{J}_\rho & \mathbf{J}_\rho^T \mathbf{J}_\mathbf{y} \\ \mathbf{J}_\mathbf{y}^T \mathbf{J}_\rho & \mathbf{J}_\mathbf{y}^T \mathbf{J}_\mathbf{y} \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho\mathbf{y}} \\ \mathbf{H}_{\mathbf{y}\rho} & \mathbf{H}_{\mathbf{y}\mathbf{y}} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x}_\rho \\ \Delta \mathbf{x}_\mathbf{y} \end{bmatrix} = \begin{bmatrix} -\mathbf{b}_\rho \\ -\mathbf{b}_\mathbf{y} \end{bmatrix}$$

$$\mathbf{J}_\rho^T \mathbf{J}_\rho = \begin{bmatrix} \frac{\partial \mathbf{r}^{(1)}}{\partial \rho^{(1)}} & \cdots & \frac{\partial \mathbf{r}^{(1)}}{\partial \rho^{(M)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathbf{r}^{(N)}}{\partial \rho^{(1)}} & \cdots & \frac{\partial \mathbf{r}^{(N)}}{\partial \rho^{(M)}} \end{bmatrix}^T \begin{bmatrix} \frac{\partial \mathbf{r}^{(1)}}{\partial \rho^{(1)}} & \cdots & \frac{\partial \mathbf{r}^{(1)}}{\partial \rho^{(M)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathbf{r}^{(N)}}{\partial \rho^{(1)}} & \cdots & \frac{\partial \mathbf{r}^{(N)}}{\partial \rho^{(M)}} \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \rho^{(1)}}^T \frac{\partial \mathbf{r}^{(i)}}{\partial \rho^{(1)}} & 0 & \cdots & 0 \\ 0 & \sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \rho^{(2)}}^T \frac{\partial \mathbf{r}^{(i)}}{\partial \rho^{(2)}} & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \cdots & \sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \rho^{(M)}}^T \frac{\partial \mathbf{r}^{(i)}}{\partial \rho^{(M)}} \end{bmatrix}$$

$$\mathbf{J}_\mathbf{y}^T \mathbf{J}_\rho = [\mathbf{J}_\mathbf{c}^T \quad \mathbf{J}_{\xi'}^T]^T \mathbf{J}_\rho = [\mathbf{J}_\mathbf{c}^T \mathbf{J}_\rho \quad \mathbf{J}_{\xi'}^T \mathbf{J}_\rho]^T$$

4.3.6. Keyframes → Sliding Window Optimization → Solving The Incremental Equation

$$\mathbf{y} = [\delta\xi^T \ a \ b \ \mathbf{c}]$$

\mathbf{H} , \mathbf{b} 행렬에 대해 자세하게 표현하면 다음과 같다.

$$\begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho\mathbf{y}} \\ \mathbf{H}_{\mathbf{y}\rho} & \mathbf{H}_{\mathbf{yy}} \end{bmatrix} = \begin{bmatrix} \mathbf{J}_\rho^T \mathbf{J}_\rho & \mathbf{J}_\rho^T \mathbf{J}_\mathbf{y} \\ \mathbf{J}_\mathbf{y}^T \mathbf{J}_\rho & \mathbf{J}_\mathbf{y}^T \mathbf{J}_\mathbf{y} \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho\mathbf{y}} \\ \mathbf{H}_{\mathbf{y}\rho} & \mathbf{H}_{\mathbf{yy}} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x}_\rho \\ \Delta \mathbf{x}_\mathbf{y} \end{bmatrix} = \begin{bmatrix} -\mathbf{b}_\rho \\ -\mathbf{b}_\mathbf{y} \end{bmatrix}$$

$$\mathbf{J}_\rho^T \mathbf{J}_\rho = \begin{bmatrix} \frac{\partial \mathbf{r}^{(1)}}{\partial \rho^{(1)}} & \dots & \frac{\partial \mathbf{r}^{(1)}}{\partial \rho^{(M)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathbf{r}^{(N)}}{\partial \rho^{(1)}} & \dots & \frac{\partial \mathbf{r}^{(N)}}{\partial \rho^{(M)}} \end{bmatrix}^T \begin{bmatrix} \frac{\partial \mathbf{r}^{(1)}}{\partial \rho^{(1)}} & \dots & \frac{\partial \mathbf{r}^{(1)}}{\partial \rho^{(M)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathbf{r}^{(N)}}{\partial \rho^{(1)}} & \dots & \frac{\partial \mathbf{r}^{(N)}}{\partial \rho^{(M)}} \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \rho^{(1)}}^T \frac{\partial \mathbf{r}^{(i)}}{\partial \rho^{(1)}} & 0 & \dots & 0 \\ 0 & \sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \rho^{(2)}}^T \frac{\partial \mathbf{r}^{(i)}}{\partial \rho^{(2)}} & \dots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \dots & \sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \rho^{(M)}}^T \frac{\partial \mathbf{r}^{(i)}}{\partial \rho^{(M)}} \end{bmatrix}$$

$$\mathbf{J}_\mathbf{y}^T \mathbf{J}_\rho = [\mathbf{J}_\mathbf{c}^T \quad \mathbf{J}_{\xi'}^T]^T \mathbf{J}_\rho = [\mathbf{J}_\mathbf{c}^T \mathbf{J}_\rho \quad \mathbf{J}_{\xi'}^T \mathbf{J}_\rho]^T$$

$$\mathbf{J}_\mathbf{c}^T \mathbf{J}_\rho = \left[\frac{\partial \mathbf{r}^{(1)}}{\partial \mathbf{c}}^T \quad \dots \quad \frac{\partial \mathbf{r}^{(N)}}{\partial \mathbf{c}}^T \right] \begin{bmatrix} \frac{\partial \mathbf{r}^{(1)}}{\partial \rho^{(1)}} & \dots & \frac{\partial \mathbf{r}^{(1)}}{\partial \rho^{(M)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathbf{r}^{(N)}}{\partial \rho^{(1)}} & \dots & \frac{\partial \mathbf{r}^{(N)}}{\partial \rho^{(M)}} \end{bmatrix} = \left[\sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \mathbf{c}}^T \frac{\partial \mathbf{r}^{(i)}}{\partial \rho^{(1)}} \quad \dots \quad \sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \mathbf{c}}^T \frac{\partial \mathbf{r}^{(i)}}{\partial \rho^{(M)}} \right]$$

4.3.6. Keyframes → Sliding Window Optimization → Solving The Incremental Equation

$$\mathbf{y} = [\delta\xi^T \ a \ b \ \mathbf{c}]$$

\mathbf{H} , \mathbf{b} 행렬에 대해 자세하게 표현하면 다음과 같다.

$$\begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho\mathbf{y}} \\ \mathbf{H}_{\mathbf{y}\rho} & \mathbf{H}_{\mathbf{y}\mathbf{y}} \end{bmatrix} = \begin{bmatrix} \mathbf{J}_\rho^T \mathbf{J}_\rho & \mathbf{J}_\rho^T \mathbf{J}_\mathbf{y} \\ \mathbf{J}_\mathbf{y}^T \mathbf{J}_\rho & \mathbf{J}_\mathbf{y}^T \mathbf{J}_\mathbf{y} \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho\mathbf{y}} \\ \mathbf{H}_{\mathbf{y}\rho} & \mathbf{H}_{\mathbf{y}\mathbf{y}} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x}_\rho \\ \Delta \mathbf{x}_\mathbf{y} \end{bmatrix} = \begin{bmatrix} -\mathbf{b}_\rho \\ -\mathbf{b}_\mathbf{y} \end{bmatrix}$$

$$\mathbf{J}_\rho^T \mathbf{J}_\rho = \begin{bmatrix} \frac{\partial \mathbf{r}^{(1)}}{\partial \rho^{(1)}} & \dots & \frac{\partial \mathbf{r}^{(1)}}{\partial \rho^{(M)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathbf{r}^{(N)}}{\partial \rho^{(1)}} & \dots & \frac{\partial \mathbf{r}^{(N)}}{\partial \rho^{(M)}} \end{bmatrix}^T \begin{bmatrix} \frac{\partial \mathbf{r}^{(1)}}{\partial \rho^{(1)}} & \dots & \frac{\partial \mathbf{r}^{(1)}}{\partial \rho^{(M)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathbf{r}^{(N)}}{\partial \rho^{(1)}} & \dots & \frac{\partial \mathbf{r}^{(N)}}{\partial \rho^{(M)}} \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \rho^{(1)}}^T \frac{\partial \mathbf{r}^{(i)}}{\partial \rho^{(1)}} & 0 & \dots & 0 \\ 0 & \sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \rho^{(2)}}^T \frac{\partial \mathbf{r}^{(i)}}{\partial \rho^{(2)}} & \dots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \dots & \sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \rho^{(M)}}^T \frac{\partial \mathbf{r}^{(i)}}{\partial \rho^{(M)}} \end{bmatrix}$$

$$\mathbf{J}_\mathbf{y}^T \mathbf{J}_\rho = [\mathbf{J}_\mathbf{c}^T \quad \mathbf{J}_{\xi'}^T]^T \mathbf{J}_\rho = [\mathbf{J}_\mathbf{c}^T \mathbf{J}_\rho \quad \mathbf{J}_{\xi'}^T \mathbf{J}_\rho]^T$$

$$\mathbf{J}_\mathbf{c}^T \mathbf{J}_\rho = \left[\frac{\partial \mathbf{r}^{(1)}}{\partial \mathbf{c}}^T \quad \dots \quad \frac{\partial \mathbf{r}^{(N)}}{\partial \mathbf{c}}^T \right] \begin{bmatrix} \frac{\partial \mathbf{r}^{(1)}}{\partial \rho^{(1)}} & \dots & \frac{\partial \mathbf{r}^{(1)}}{\partial \rho^{(M)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathbf{r}^{(N)}}{\partial \rho^{(1)}} & \dots & \frac{\partial \mathbf{r}^{(N)}}{\partial \rho^{(M)}} \end{bmatrix} = \left[\sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \mathbf{c}}^T \frac{\partial \mathbf{r}^{(i)}}{\partial \rho^{(1)}} \quad \dots \quad \sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \mathbf{c}}^T \frac{\partial \mathbf{r}^{(i)}}{\partial \rho^{(M)}} \right]$$

$$\mathbf{J}_{\xi'}^T \mathbf{J}_\rho = \begin{bmatrix} \frac{\partial \mathbf{r}^{(1)}}{\partial \xi_1} & \dots & \frac{\partial \mathbf{r}^{(1)}}{\partial \xi_8} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathbf{r}^{(N)}}{\partial \xi_1} & \dots & \frac{\partial \mathbf{r}^{(N)}}{\partial \xi_8} \end{bmatrix}^T \begin{bmatrix} \frac{\partial \mathbf{r}^{(1)}}{\partial \rho^{(1)}} & \dots & \frac{\partial \mathbf{r}^{(1)}}{\partial \rho^{(M)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathbf{r}^{(N)}}{\partial \rho^{(1)}} & \dots & \frac{\partial \mathbf{r}^{(N)}}{\partial \rho^{(M)}} \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \xi_1}^T \frac{\partial \mathbf{r}^{(i)}}{\partial \rho^{(1)}} & \dots & \sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \xi_1}^T \frac{\partial \mathbf{r}^{(i)}}{\partial \rho^{(M)}} \\ \vdots & \ddots & \vdots \\ \sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \xi_8}^T \frac{\partial \mathbf{r}^{(i)}}{\partial \rho^{(1)}} & \dots & \sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \xi_8}^T \frac{\partial \mathbf{r}^{(i)}}{\partial \rho^{(M)}} \end{bmatrix}$$

more detail: <https://www.cnblogs.com/jingeTU/p/8395046.html>

4.3.6. Keyframes → Sliding Window Optimization → Solving The Incremental Equation

H, b 행렬에 대해 자세하게 표현하면 다음과 같다.

$$\begin{bmatrix} H_{\rho\rho} & H_{\rho y} \\ H_{y\rho} & H_{yy} \end{bmatrix} \begin{bmatrix} \Delta x_\rho \\ \Delta x_y \end{bmatrix} = \begin{bmatrix} -b_\rho \\ -b_y \end{bmatrix}$$

$f(\mathbf{x}) = \sqrt{w_h} \mathbf{r}$ 이므로 huber norm을 생략하고 residual만 표현하면

$$\begin{bmatrix} b_\rho \\ b_y \end{bmatrix} \xrightarrow{\quad} \begin{bmatrix} \mathbf{J}_\rho^T f(\mathbf{x}) \\ \mathbf{J}_y^T f(\mathbf{x}) \end{bmatrix} \xrightarrow{\quad} \sqrt{w_h} \begin{bmatrix} \mathbf{J}_\rho^T \mathbf{r} \\ \mathbf{J}_y^T \mathbf{r} \end{bmatrix}$$

4.3.6. Keyframes → Sliding Window Optimization → Solving The Incremental Equation

H, b 행렬에 대해 자세하게 표현하면 다음과 같다.

$$\begin{bmatrix} H_{\rho\rho} & H_{\rho y} \\ H_{y\rho} & H_{yy} \end{bmatrix} \begin{bmatrix} \Delta x_\rho \\ \Delta x_y \end{bmatrix} = \begin{bmatrix} -b_\rho \\ -b_y \end{bmatrix}$$

$f(\mathbf{x}) = \sqrt{w_h} \mathbf{r}$ 이므로 huber norm을 생략하고 residual만 표현하면

$$\begin{bmatrix} b_\rho \\ b_y \end{bmatrix} \rightarrow \begin{bmatrix} \mathbf{J}_\rho^T f(\mathbf{x}) \\ \mathbf{J}_y^T f(\mathbf{x}) \end{bmatrix} \rightarrow \sqrt{w_h} \begin{bmatrix} \mathbf{J}_\rho^T \mathbf{r} \\ \mathbf{J}_y^T \mathbf{r} \end{bmatrix}$$

$$\mathbf{J}_\rho^T \mathbf{r} = \begin{bmatrix} \frac{\partial \mathbf{r}^{(1)}}{\partial \rho^{(1)}} & \cdots & \frac{\partial \mathbf{r}^{(1)}}{\partial \rho^{(M)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathbf{r}^{(N)}}{\partial \rho^{(1)}} & \cdots & \frac{\partial \mathbf{r}^{(N)}}{\partial \rho^{(M)}} \end{bmatrix}^T \begin{bmatrix} \mathbf{r}^{(1)} \\ \vdots \\ \mathbf{r}^{(N)} \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \rho^{(1)}}^T \mathbf{r}^{(i)} \\ \vdots \\ \sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \rho^{(M)}}^T \mathbf{r}^{(i)} \end{bmatrix}$$

4.3.6. Keyframes → Sliding Window Optimization → Solving The Incremental Equation

H, b 행렬에 대해 자세하게 표현하면 다음과 같다.

$$\begin{bmatrix} H_{\rho\rho} & H_{\rho y} \\ H_{y\rho} & H_{yy} \end{bmatrix} \begin{bmatrix} \Delta x_\rho \\ \Delta x_y \end{bmatrix} = \begin{bmatrix} -b_\rho \\ -b_y \end{bmatrix}$$

$f(\mathbf{x}) = \sqrt{w_h} \mathbf{r}$ 이므로 huber norm을 생략하고 residual만 표현하면

$$\begin{bmatrix} b_\rho \\ b_y \end{bmatrix} \rightarrow \begin{bmatrix} J_\rho^T f(\mathbf{x}) \\ J_y^T f(\mathbf{x}) \end{bmatrix} \rightarrow \sqrt{w_h} \begin{bmatrix} J_\rho^T \mathbf{r} \\ J_y^T \mathbf{r} \end{bmatrix}$$

$$J_\rho^T \mathbf{r} = \begin{bmatrix} \frac{\partial \mathbf{r}^{(1)}}{\partial \rho^{(1)}} & \cdots & \frac{\partial \mathbf{r}^{(1)}}{\partial \rho^{(M)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathbf{r}^{(N)}}{\partial \rho^{(1)}} & \cdots & \frac{\partial \mathbf{r}^{(N)}}{\partial \rho^{(M)}} \end{bmatrix}^T \begin{bmatrix} \mathbf{r}^{(1)} \\ \vdots \\ \mathbf{r}^{(N)} \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \rho^{(1)}}^T \mathbf{r}^{(i)} \\ \vdots \\ \sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \rho^{(M)}}^T \mathbf{r}^{(i)} \end{bmatrix}$$

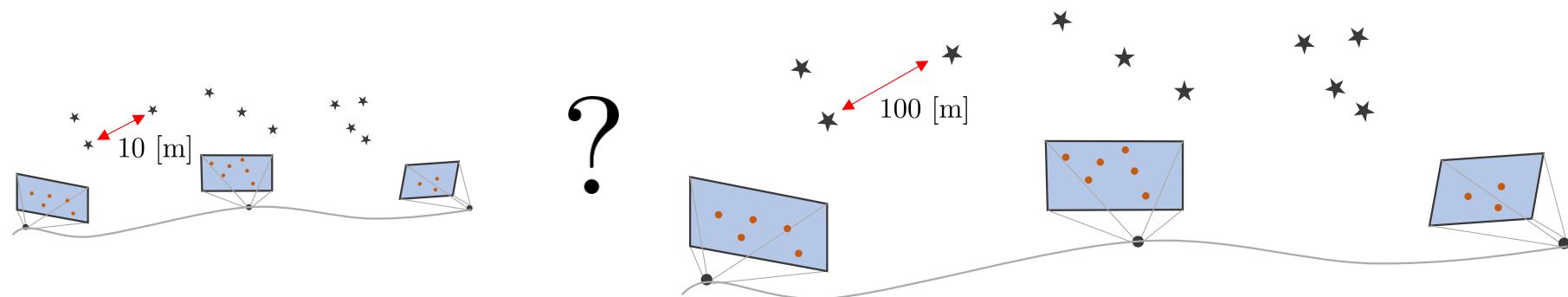
$$J_y^T \mathbf{r} = \begin{bmatrix} J_c^T \\ J_{\xi'}^T \end{bmatrix} \mathbf{r} = \begin{bmatrix} \frac{\partial \mathbf{r}^{(1)}}{\partial \mathbf{c}}^T & \cdots & \frac{\partial \mathbf{r}^{(1)}}{\partial \mathbf{c}}^T \\ \frac{\partial \mathbf{r}^{(1)}}{\partial \xi_1}^T & \cdots & \frac{\partial \mathbf{r}^{(1)}}{\partial \xi_8}^T \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathbf{r}^{(N)}}{\partial \xi_1}^T & \cdots & \frac{\partial \mathbf{r}^{(N)}}{\partial \xi_8}^T \end{bmatrix} \begin{bmatrix} \mathbf{r}^{(1)} \\ \vdots \\ \mathbf{r}^{(N)} \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \mathbf{c}}^T \mathbf{r}^{(i)} \\ \sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \xi_1}^T \mathbf{r}^{(i)} \\ \vdots \\ \sum_{i=1}^N \frac{\partial \mathbf{r}^{(i)}}{\partial \xi_8}^T \mathbf{r}^{(i)} \end{bmatrix}$$

4.4. Null Space Effect Elimination

Ambiguity Problems

DSO는 단안카메라를 활용한 Visual Odometry이므로 Map Point의 깊이를 정확하게 알 수 없는 Scale Ambiguity 문제를 가진다.

예를 들면, 내가 현재 실제 도시의 이미지를 보고 있는 것인지 도시의 미니어쳐 이미지를 보고 있는 것인지를 단안카메라만 사용해서는 정확하게 알 수 없다는 의미이다



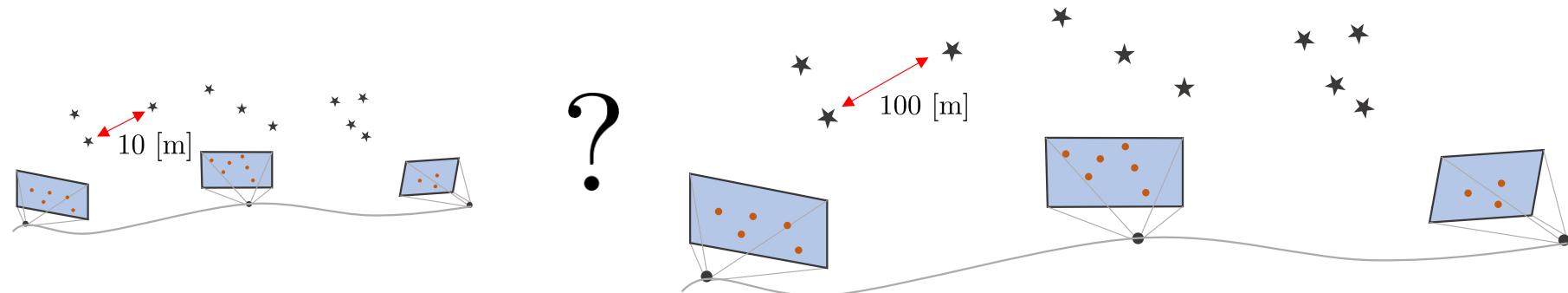
more detail: <http://www.lingtonq.de/2020/04/24/DSO-Null-Space/>
<https://blog.csdn.net/wubaobao1993/article/details/105106301>

4.4. Null Space Effect Elimination

Ambiguity Problems

DSO는 단안카메라를 활용한 Visual Odometry이므로 Map Point의 깊이를 정확하게 알 수 없는 Scale Ambiguity 문제를 가진다.

예를 들면, 내가 현재 실제 도시의 이미지를 보고 있는 것인지 도시의 미니어쳐 이미지를 보고 있는 것인지를 단안카메라만 사용해서는 정확하게 알 수 없다는 의미이다
즉, 카메라의 Trajectory와 모든 Map Point들을 일괄적으로 키우거나 줄여도 최적화 단계에서 아무런 영향을 미치지 않는다



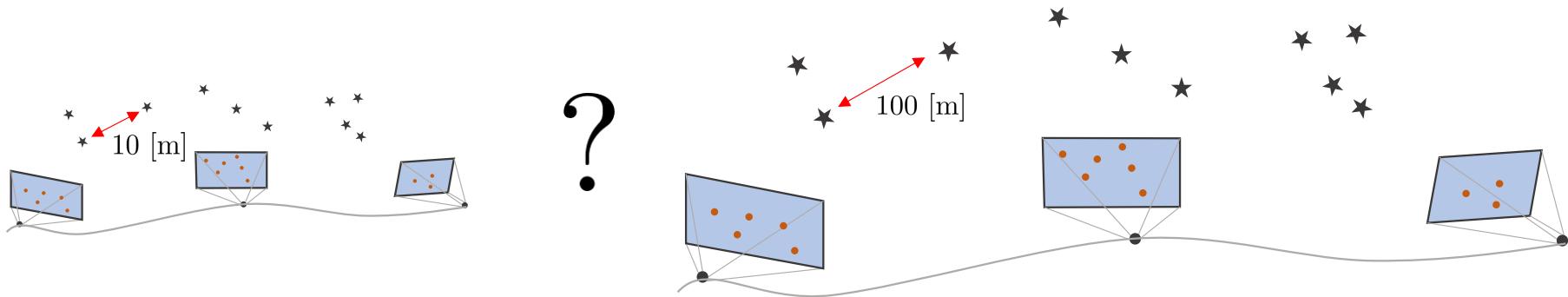
more detail: <http://www.lingtonq.de/2020/04/24/DSO-Null-Space/>
<https://blog.csdn.net/wubaobao1993/article/details/105106301>

4.4. Null Space Effect Elimination

Ambiguity Problems

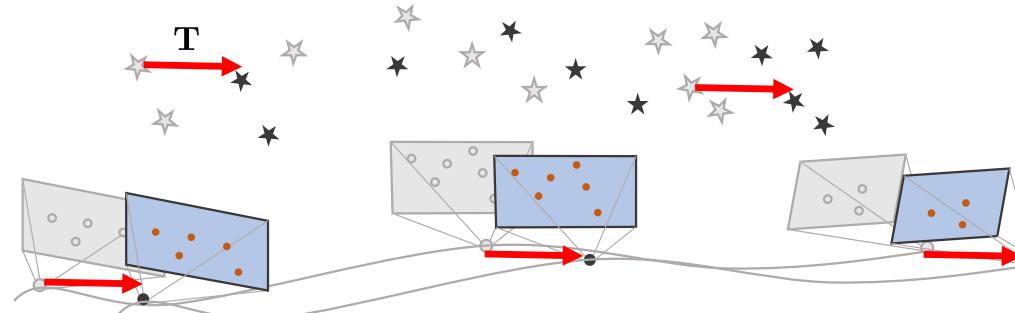
DSO는 단안카메라를 활용한 Visual Odometry이므로 Map Point의 깊이를 정확하게 알 수 없는 Scale Ambiguity 문제를 가진다.

예를 들면, 내가 현재 실제 도시의 이미지를 보고 있는 것인지 도시의 미니어쳐 이미지를 보고 있는 것인지를 단안카메라만 사용해서는 정확하게 알 수 없다는 의미이다 즉, 카메라의 Trajectory와 모든 Map Point들을 일괄적으로 키우거나 줄여도 최적화 단계에서 아무런 영향을 미치지 않는다



또한 모든 카메라 포즈들과 Map Point에 동일한 SE3 변환을 적용하면 이 또한 최적화 단계에서 아무런 영향을 미치지 않는다

이를 Coordinate System Ambiguity 문제라고 한다



more detail: <http://www.lingtonq.de/2020/04/24/DSO-Null-Space/>
<https://blog.csdn.net/wubaobao1993/article/details/105106301>

4.4. Null Space Effect Elimination

Null Space

Null Space란 정방행렬 $\mathbf{A} \in \mathbb{R}^{n \times n}$ 과 벡터 $\mathbf{x} \in \mathbb{R}^n$ 가 존재할 때 다음을 만족하는 해집합을 의미한다.

$$\mathbf{Ax} = 0$$

more detail: <http://www.lingtonq.de/2020/04/24/DSO-Null-Space/>
<https://blog.csdn.net/wubaobao1993/article/details/105106301>

4.4. Null Space Effect Elimination

Null Space

Null Space란 정방행렬 $\mathbf{A} \in \mathbb{R}^{n \times n}$ 과 벡터 $\mathbf{x} \in \mathbb{R}^n$ 가 존재할 때 다음을 만족하는 해집합을 의미한다.

$$\mathbf{Ax} = 0$$

일반적으로 Gauss-Newton 방법을 사용할 때 다음과 같은 공식을 풀어야 하는 상황이 발생한다.

$$\mathbf{H}\Delta\mathbf{x} = -\mathbf{b}$$

where, $\mathbf{H} = \mathbf{J}^T \mathbf{J} \in \mathbb{R}^{n \times n}$

$$\mathbf{b} = \mathbf{J}^T \mathbf{e} \in \mathbb{R}^n$$

$$\Delta\mathbf{x} \in \mathbb{R}^n$$

more detail: <http://www.lingtonq.de/2020/04/24/DSO-Null-Space/>
<https://blog.csdn.net/wubaobao1993/article/details/105106301>

4.4. Null Space Effect Elimination

Null Space

Null Space란 정방행렬 $\mathbf{A} \in \mathbb{R}^{n \times n}$ 과 벡터 $\mathbf{x} \in \mathbb{R}^n$ 가 존재할 때 다음을 만족하는 해집합을 의미한다.

$$\mathbf{Ax} = 0$$

일반적으로 Gauss-Newton 방법을 사용할 때 다음과 같은 공식을 풀어야 하는 상황이 발생한다.

$$\mathbf{H}\Delta\mathbf{x} = -\mathbf{b}$$

where, $\mathbf{H} = \mathbf{J}^T \mathbf{J} \in \mathbb{R}^{n \times n}$

$$\mathbf{b} = \mathbf{J}^T \mathbf{e} \in \mathbb{R}^n$$

$$\Delta\mathbf{x} \in \mathbb{R}^n$$

만약 \mathbf{H} 의 역행렬이 존재하는 경우 $\Delta\mathbf{x}$ 는 다음과 같은 유일해(Unique Solution)을 가진다.

$$\Delta\mathbf{x} = -\mathbf{H}^{-1}\mathbf{b} \quad \text{if } \mathbf{H} \text{ is invertible.}$$

more detail: <http://www.lingtonq.de/2020/04/24/DSO-Null-Space/>
<https://blog.csdn.net/wubaobao1993/article/details/105106301>

4.4. Null Space Effect Elimination

Null Space

Null Space란 정방행렬 $\mathbf{A} \in \mathbb{R}^{n \times n}$ 과 벡터 $\mathbf{x} \in \mathbb{R}^n$ 가 존재할 때 다음을 만족하는 해집합을 의미한다.

$$\mathbf{Ax} = 0$$

일반적으로 Gauss-Newton 방법을 사용할 때 다음과 같은 공식을 풀어야 하는 상황이 발생한다.

$$\mathbf{H}\Delta\mathbf{x} = -\mathbf{b}$$

where, $\mathbf{H} = \mathbf{J}^T \mathbf{J} \in \mathbb{R}^{n \times n}$

$$\mathbf{b} = \mathbf{J}^T \mathbf{e} \in \mathbb{R}^n$$

$$\Delta\mathbf{x} \in \mathbb{R}^n$$

만약 \mathbf{H} 의 역행렬이 존재하는 경우 $\Delta\mathbf{x}$ 는 다음과 같은 유일해(Unique Solution)을 가진다.

$$\Delta\mathbf{x} = -\mathbf{H}^{-1}\mathbf{b} \quad \text{if } \mathbf{H} \text{ is invertible.}$$

만약 \mathbf{H} 의 역행렬이 존재하지 않는 경우 $\Delta\mathbf{x}$ 는 특수해(Particular Solution) $\Delta\mathbf{x}_p$ 와 일반해(Homogeneous Solution) $\Delta\mathbf{x}_h$ 를 합한 완전해(Complete Solution)을 갖는다.

$$\Delta\mathbf{x} = \Delta\mathbf{x}_h + \Delta\mathbf{x}_p \quad \text{if } \mathbf{H} \text{ is not invertible.}$$

more detail: <http://www.lingtonq.de/2020/04/24/DSO-Null-Space/>
<https://blog.csdn.net/wubaobao1993/article/details/105106301>

4.4. Null Space Effect Elimination

Null Space

우선 비동차방정식(Non-Homogeneous Equation)에서 특수해 $\Delta\mathbf{x}_p$ 를 구할 수 있다.

$$\mathbf{H}\Delta\mathbf{x}_p = -\mathbf{b}$$

more detail: <http://www.lingtonq.de/2020/04/24/DSO-Null-Space/>
<https://blog.csdn.net/wubaobao1993/article/details/105106301>

4.4. Null Space Effect Elimination

Null Space

우선 비동차방정식(Non-Homogeneous Equation)에서 특수해 $\Delta\mathbf{x}_p$ 를 구할 수 있다.

$$\mathbf{H}\Delta\mathbf{x}_p = -\mathbf{b}$$

다음으로 동차방정식(Homogeneous Equation)에서 일반해 $\Delta\mathbf{x}_h$ 를 구할 수 있다.

$$\mathbf{H}\Delta\mathbf{x}_h = 0$$

more detail: <http://www.lingtonq.de/2020/04/24/DSO-Null-Space/>
<https://blog.csdn.net/wubaobao1993/article/details/105106301>

4.4. Null Space Effect Elimination

Null Space

우선 비동차방정식(Non-Homogeneous Equation)에서 특수해 $\Delta\mathbf{x}_p$ 를 구할 수 있다.

$$\mathbf{H}\Delta\mathbf{x}_p = -\mathbf{b}$$

다음으로 동차방정식(Homogeneous Equation)에서 일반해 $\Delta\mathbf{x}_h$ 를 구할 수 있다.

$$\mathbf{H}\Delta\mathbf{x}_h = 0$$

따라서 \mathbf{H} 의 역행렬이 존재하지 않는 경우 완전해는 다음과 같이 나타낼 수 있다.

이 때 λ 는 임의의 정수를 의미하며 λ 값에 관계없이 아래 식은 항상 성립하므로 완전해는 무수히 많은 해를 가진다.

$$\Delta\mathbf{x} = \Delta\mathbf{x}_p + \lambda\Delta\mathbf{x}_h$$

$$\mathbf{H}(\Delta\mathbf{x}_p + \lambda\Delta\mathbf{x}_h) = -\mathbf{b}$$

more detail: <http://www.lingtonq.de/2020/04/24/DSO-Null-Space/>
<https://blog.csdn.net/wubaobao1993/article/details/105106301>

4.4. Null Space Effect Elimination

Null Space

우선 비동차방정식(Non-Homogeneous Equation)에서 특수해 $\Delta\mathbf{x}_p$ 를 구할 수 있다.

$$\mathbf{H}\Delta\mathbf{x}_p = -\mathbf{b}$$

다음으로 동차방정식(Homogeneous Equation)에서 일반해 $\Delta\mathbf{x}_h$ 를 구할 수 있다.

$$\mathbf{H}\Delta\mathbf{x}_h = 0$$

따라서 \mathbf{H} 의 역행렬이 존재하지 않는 경우 완전해는 다음과 같이 나타낼 수 있다.

이 때 λ 는 임의의 정수를 의미하며 λ 값에 관계없이 아래 식은 항상 성립하므로 완전해는 무수히 많은 해를 가진다.

$$\Delta\mathbf{x} = \Delta\mathbf{x}_p + \lambda\Delta\mathbf{x}_h$$

$$\mathbf{H}(\Delta\mathbf{x}_p + \lambda\Delta\mathbf{x}_h) = -\mathbf{b}$$

여기서 Null Space는 $\Delta\mathbf{x}_h$ 의 해집합으로 인해 Span되는 공간을 의미한다.

$$\mathbf{N} = \text{Span}(\Delta\mathbf{x}_h)$$

more detail: <http://www.lingtonq.de/2020/04/24/DSO-Null-Space/>
<https://blog.csdn.net/wubaobao1993/article/details/105106301>

4.4. Null Space Effect Elimination

Null Space

우선 비동차방정식(Non-Homogeneous Equation)에서 특수해 $\Delta\mathbf{x}_p$ 를 구할 수 있다.

$$\mathbf{H}\Delta\mathbf{x}_p = -\mathbf{b}$$

다음으로 동차방정식(Homogeneous Equation)에서 일반해 $\Delta\mathbf{x}_h$ 를 구할 수 있다.

$$\mathbf{H}\Delta\mathbf{x}_h = 0$$

따라서 \mathbf{H} 의 역행렬이 존재하지 않는 경우 완전해는 다음과 같이 나타낼 수 있다.

이 때 λ 는 임의의 정수를 의미하며 λ 값에 관계없이 아래 식은 항상 성립하므로 완전해는 무수히 많은 해를 가진다.

$$\Delta\mathbf{x} = \Delta\mathbf{x}_p + \lambda\Delta\mathbf{x}_h$$

$$\mathbf{H}(\Delta\mathbf{x}_p + \lambda\Delta\mathbf{x}_h) = -\mathbf{b}$$

여기서 Null Space는 $\Delta\mathbf{x}_h$ 의 해집합으로 인해 Span되는 공간을 의미한다.

$$\mathbf{N} = \text{Span}(\Delta\mathbf{x}_h)$$

Null Space는 전체 방정식의 값에 영향을 미치지 않으므로 Null Space의 선형결합인 $\mathbf{N}\Delta\mathbf{x}$ 또한 Null Space에 포함되어 다음과 같은 방정식이 성립한다.

$$\mathbf{H}\Delta\mathbf{x} + \mathbf{N}\Delta\mathbf{x} = -\mathbf{b}$$

more detail: <http://www.lingtonq.de/2020/04/24/DSO-Null-Space/>
<https://blog.csdn.net/wubaobao1993/article/details/105106301>

4.4. Null Space Effect Elimination

Null Space in DSO

Null Space는 시스템(=에러 함수) $\|E(x + \Delta x)\|$ 값에는 영향을 미치지 않지만 업데이트 값 Δx 에는 영향을 미치므로 Null Space의 영향이 곧 Ambiguity 문제의 원인이 된다 따라서 DSO에서는 LBA를 수행할 때 Null Space의 영향을 제거하여 Ambiguity 문제를 해결하는 방법을 사용했다

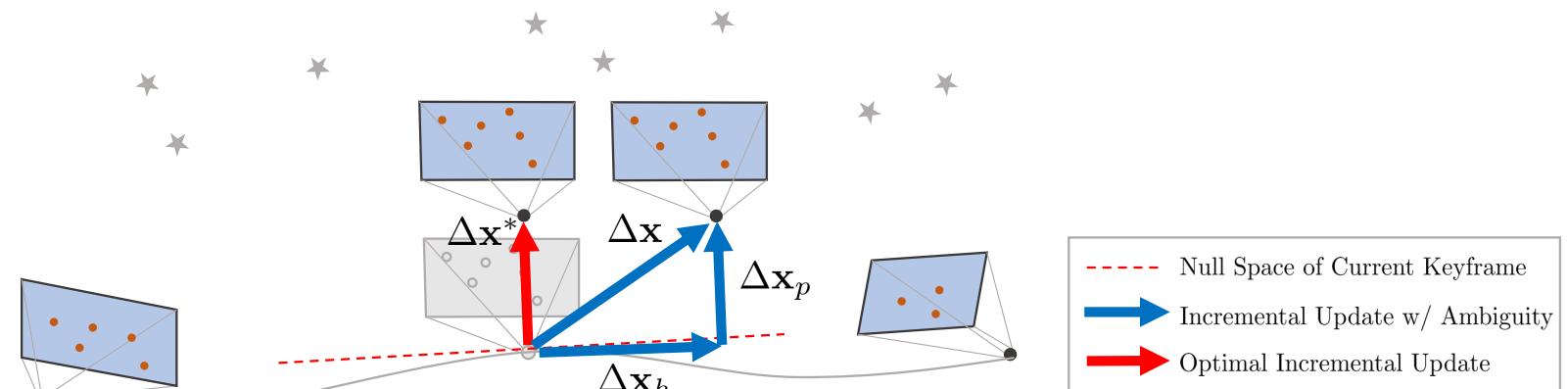


Figure is inspired from: <https://blog.csdn.net/xxlinttp/article/details/100080080>

more detail: <http://www.lingtonq.de/2020/04/24/DSO-Null-Space/>
<https://blog.csdn.net/wubaobao1993/article/details/105106301>

4.4. Null Space Effect Elimination

Null Space in DSO

Null Space는 시스템(=에러 함수) $\|E(x + \Delta x)\|$ 값에는 영향을 미치지 않지만 업데이트 값 Δx 에는 영향을 미치므로 Null Space의 영향이 곧 Ambiguity 문제의 원인이 된다 따라서 DSO에서는 LBA를 수행할 때 Null Space의 영향을 제거하여 Ambiguity 문제를 해결하는 방법을 사용했다

예를 들어 아래 그림과 같이 LBA를 통해 업데이트 값 $\Delta x = \Delta x_p + \Delta x_h$ 이 구해졌다고 가정하면 여기에서 Null Space에 프로젝션된 양 $P_N \Delta x$ 만큼을 빼줌으로써 Null Space의 영향을 제거할 수 있다

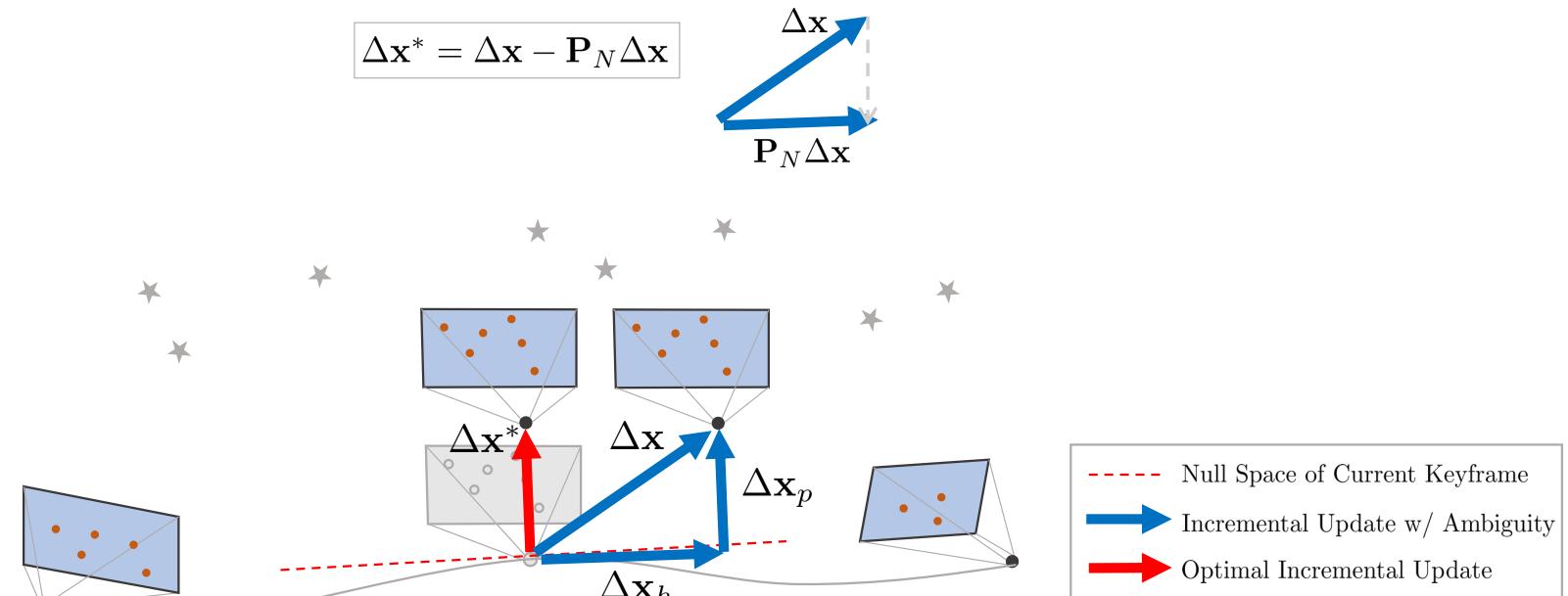


Figure is inspired from: <https://blog.csdn.net/xxlinttp/article/details/100080080>

more detail: <http://www.lingtonq.de/2020/04/24/DSO-Null-Space/>
<https://blog.csdn.net/wubaobao1993/article/details/105106301>

4.4. Null Space Effect Elimination

Null Space in DSO

프로젝션 행렬 \mathbf{P}_N 은 $\mathbf{N}^T \mathbf{N}$ 의 역행렬이 존재하는가 안하는가에 따라 2가지 방법으로 구할 수 있다.

$$\begin{aligned}\mathbf{P}_N &= \mathbf{N} \mathbf{N}^\dagger \\ &= \mathbf{N} (\mathbf{N}^T \mathbf{N})^{-1} \mathbf{N}^T \quad \text{if, } \mathbf{N} \mathbf{N}^T \text{ is invertible.} \\ &= \mathbf{N} (\mathbf{V} \Sigma^\dagger \mathbf{U}^T) \quad \text{if, } \mathbf{N} \mathbf{N}^T \text{ is not invertible. Do SVD}\end{aligned}$$

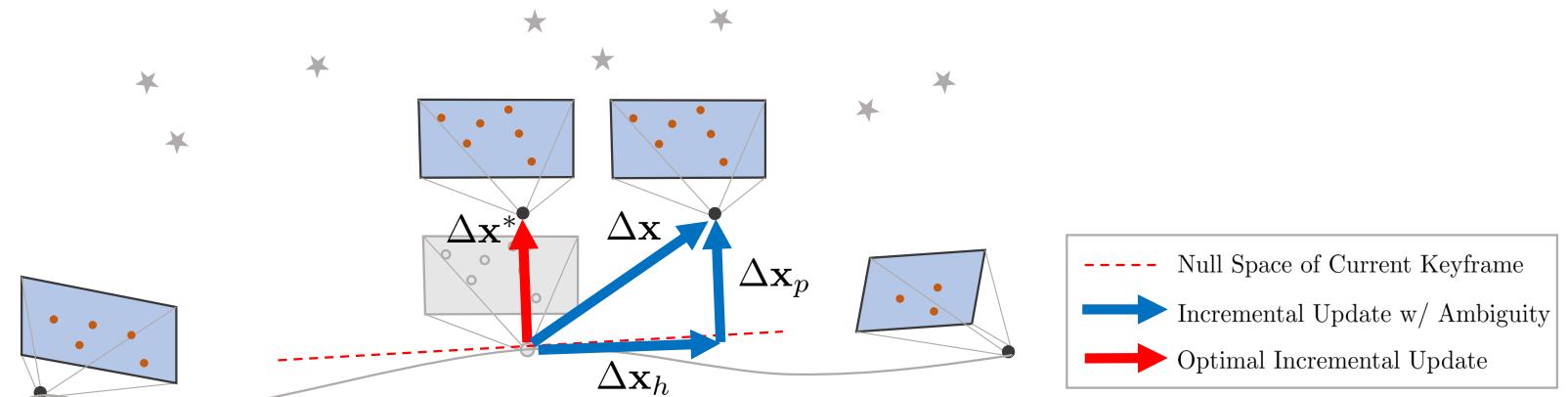


Figure is inspired from: <https://blog.csdn.net/xxlinttp/article/details/100080080>

more detail: <http://www.lingtonq.de/2020/04/24/DSO-Null-Space/>
<https://blog.csdn.net/wubaobao1993/article/details/105106301>

4.4. Null Space Effect Elimination

Null Space in DSO

프로젝션 행렬 \mathbf{P}_N 은 $\mathbf{N}^T \mathbf{N}$ 의 역행렬이 존재하는가 안하는가에 따라 2가지 방법으로 구할 수 있다.

$$\begin{aligned}\mathbf{P}_N &= \mathbf{N} \mathbf{N}^\dagger \\ &= \mathbf{N} (\mathbf{N}^T \mathbf{N})^{-1} \mathbf{N}^T \quad \text{if, } \mathbf{N} \mathbf{N}^T \text{ is invertible.} \\ &= \mathbf{N} (\mathbf{V} \Sigma^\dagger \mathbf{U}^T) \quad \text{if, } \mathbf{N} \mathbf{N}^T \text{ is not invertible. Do SVD}\end{aligned}$$

결론적으로 다음과 같이 최적의 업데이트 값 $\Delta\mathbf{x}^*$ 를 계산함으로써 Null Space의 영향을 제거할 수 있다

$$\Delta\mathbf{x}^* = \Delta\mathbf{x} - \mathbf{N} \mathbf{N}^\dagger \Delta\mathbf{x}$$

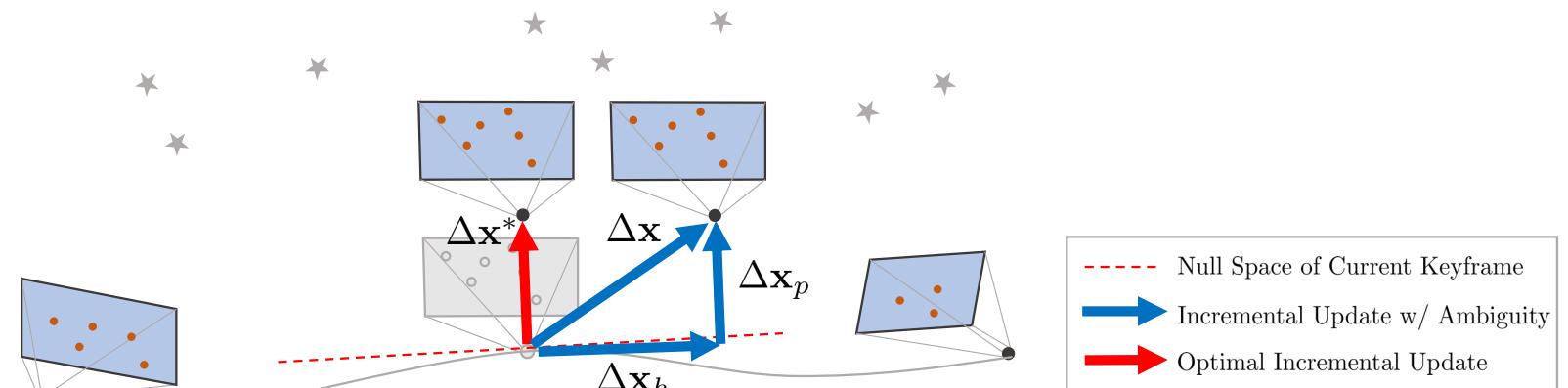


Figure is inspired from: <https://blog.csdn.net/xxlinttp/article/details/100080080>

more detail: <http://www.lingtonq.de/2020/04/24/DSO-Null-Space/>
<https://blog.csdn.net/wubaobao1993/article/details/105106301>

4.4. Null Space Effect Elimination

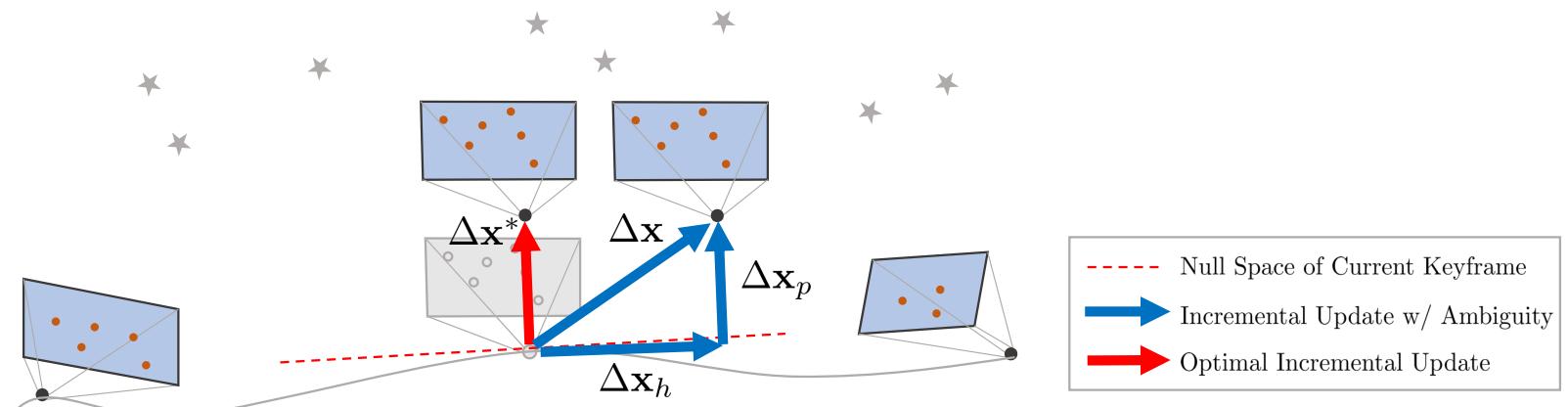
Detailed Derivation of Null Space in DSO

프로젝션 행렬을 사용하지 않고 다른 방법으로 유도하면 다음과 같다. 우선 최적의 업데이트 값 $\Delta\mathbf{x}^*$ 는 다음과 같이 나타낼 수 있다.

이 때, $\mathbf{N}\Delta\mathbf{x}_h$ 는 Null Space에 의해 드리프트된 값을 의미한다.

$$\Delta\mathbf{x}^* = \Delta\mathbf{x} - \mathbf{N}\Delta\mathbf{x}_h$$

왜 해당 공식이 성립하는지는 정확히 이해하지 못함



more detail: <http://www.lingtonq.de/2020/04/24/DSO-Null-Space/>
<https://blog.csdn.net/wubaobao1993/article/details/105106301>

4.4. Null Space Effect Elimination

Detailed Derivation of Null Space in DSO

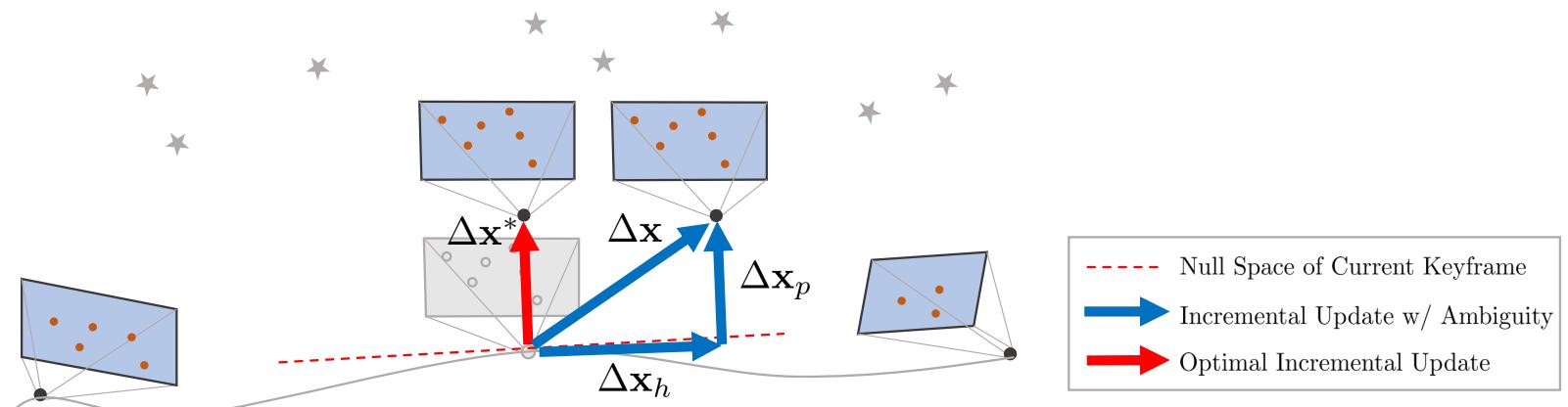
프로젝션 행렬을 사용하지 않고 다른 방법으로 유도하면 다음과 같다. 우선 최적의 업데이트 값 $\Delta\mathbf{x}^*$ 는 다음과 같이 나타낼 수 있다.

이 때, $\mathbf{N}\Delta\mathbf{x}_h$ 는 Null Space에 의해 드리프트된 값을 의미한다.

$$\Delta\mathbf{x}^* = \Delta\mathbf{x} - \mathbf{N}\Delta\mathbf{x}_h$$

Null Space가 여러 함수의 값에 영향을 미치지 않는다는 점을 사용하면 다음 공식이 성립한다.

$$\mathbf{E}(\mathbf{x} + \Delta\mathbf{x}) = \mathbf{E}(\mathbf{x} + \Delta\mathbf{x} - \mathbf{N}\Delta\mathbf{x}_h)$$



more detail: <http://www.lingtonq.de/2020/04/24/DSO-Null-Space/>
<https://blog.csdn.net/wubaobao1993/article/details/105106301>

4.4. Null Space Effect Elimination

Detailed Derivation of Null Space in DSO

$$\mathbf{E}(\mathbf{x} + \Delta\mathbf{x}) = \mathbf{E}(\mathbf{x} + \Delta\mathbf{x} - \mathbf{N}\Delta\mathbf{x}_h)$$

위 공식의 두 에러함수를 1차 테일러 전개하여 정리하면 다음과 같이 나타낼 수 있다

more detail: <http://www.lingtonq.de/2020/04/24/DSO-Null-Space/>
<https://blog.csdn.net/wubaobao1993/article/details/105106301>

4.4. Null Space Effect Elimination

Detailed Derivation of Null Space in DSO

$$\mathbf{E}(\mathbf{x} + \Delta\mathbf{x}) = \mathbf{E}(\mathbf{x} + \Delta\mathbf{x} - \mathbf{N}\Delta\mathbf{x}_h)$$

위 공식의 두 에러함수를 1차 테일러 전개하여 정리하면 다음과 같이 나타낼 수 있다

$$\begin{aligned}\mathbf{E}(\mathbf{x} + \Delta\mathbf{x} - \mathbf{N}\Delta\mathbf{x}_h) &= \mathbf{e}(\mathbf{x} + \Delta\mathbf{x} - \mathbf{N}\Delta\mathbf{x}_h)^T \mathbf{e}(\mathbf{x} + \Delta\mathbf{x} - \mathbf{N}\Delta\mathbf{x}_h) \\ &\simeq (\mathbf{e}(\mathbf{x}) + \mathbf{J}(\Delta\mathbf{x} - \mathbf{N}\Delta\mathbf{x}_h))^T (\mathbf{e}(\mathbf{x}) + \mathbf{J}(\Delta\mathbf{x} - \mathbf{N}\Delta\mathbf{x}_h)) \\ &= \underbrace{\mathbf{e}^T \mathbf{e} + \mathbf{e}^T \mathbf{J} \Delta\mathbf{x} + \Delta\mathbf{x} \mathbf{J}^T \mathbf{e} + \Delta\mathbf{x}^T \mathbf{J}^T \mathbf{J} \Delta\mathbf{x}}_{\mathbf{E}(\mathbf{x} + \Delta\mathbf{x})} - \underbrace{\mathbf{e}^T \mathbf{J} \mathbf{N} \Delta\mathbf{x}_h - (\mathbf{N} \Delta\mathbf{x}_h)^T \mathbf{J}^T \mathbf{e} - (\mathbf{N} \Delta\mathbf{x}_h)^T \mathbf{J}^T \mathbf{J} \Delta\mathbf{x} - \Delta\mathbf{x} \mathbf{J}^T \mathbf{J} (\mathbf{N} \Delta\mathbf{x}_h) + (\mathbf{N} \Delta\mathbf{x}_h)^T \mathbf{J}^T \mathbf{J} (\mathbf{N} \Delta\mathbf{x}_h)}_{\Delta\mathbf{x}_h \text{ part}} \\ &= \mathbf{E}(\mathbf{x} + \Delta\mathbf{x})\end{aligned}$$

more detail: <http://www.lingtong.de/2020/04/24/DSO-Null-Space/>
<https://blog.csdn.net/wubaobao1993/article/details/105106301>

4.4. Null Space Effect Elimination

Detailed Derivation of Null Space in DSO

$$\mathbf{E}(\mathbf{x} + \Delta\mathbf{x}) = \mathbf{E}(\mathbf{x} + \Delta\mathbf{x} - \mathbf{N}\Delta\mathbf{x}_h)$$

위 공식의 두 에러함수를 1차 테일러 전개하여 정리하면 다음과 같이 나타낼 수 있다

$$\begin{aligned}\mathbf{E}(\mathbf{x} + \Delta\mathbf{x} - \mathbf{N}\Delta\mathbf{x}_h) &= \mathbf{e}(\mathbf{x} + \Delta\mathbf{x} - \mathbf{N}\Delta\mathbf{x}_h)^T \mathbf{e}(\mathbf{x} + \Delta\mathbf{x} - \mathbf{N}\Delta\mathbf{x}_h) \\ &\simeq (\mathbf{e}(\mathbf{x}) + \mathbf{J}(\Delta\mathbf{x} - \mathbf{N}\Delta\mathbf{x}_h))^T (\mathbf{e}(\mathbf{x}) + \mathbf{J}(\Delta\mathbf{x} - \mathbf{N}\Delta\mathbf{x}_h)) \\ &= \underbrace{\mathbf{e}^T \mathbf{e} + \mathbf{e}^T \mathbf{J} \Delta\mathbf{x} + \Delta\mathbf{x} \mathbf{J}^T \mathbf{e} + \Delta\mathbf{x}^T \mathbf{J}^T \mathbf{J} \Delta\mathbf{x}}_{\mathbf{E}(\mathbf{x} + \Delta\mathbf{x})} - \underbrace{\mathbf{e}^T \mathbf{J} \mathbf{N} \Delta\mathbf{x}_h - (\mathbf{N} \Delta\mathbf{x}_h)^T \mathbf{J}^T \mathbf{e} - (\mathbf{N} \Delta\mathbf{x}_h)^T \mathbf{J}^T \mathbf{J} \Delta\mathbf{x} - \Delta\mathbf{x} \mathbf{J}^T \mathbf{J} (\mathbf{N} \Delta\mathbf{x}_h) + (\mathbf{N} \Delta\mathbf{x}_h)^T \mathbf{J}^T \mathbf{J} (\mathbf{N} \Delta\mathbf{x}_h)}_{\Delta\mathbf{x}_h \text{ part}} \\ &= \mathbf{E}(\mathbf{x} + \Delta\mathbf{x})\end{aligned}$$

따라서 공통부분을 소거한 후 다시 정리하면 다음과 같은 공식이 성립하고

$$(\mathbf{N} \Delta\mathbf{x}_h)^T \mathbf{J}^T \mathbf{J} (\mathbf{N} \Delta\mathbf{x}_h) = \mathbf{e}^T \mathbf{J} \mathbf{N} \Delta\mathbf{x}_h + (\mathbf{N} \Delta\mathbf{x}_h)^T \mathbf{J}^T \mathbf{e} + (\mathbf{N} \Delta\mathbf{x}_h)^T \mathbf{J}^T \mathbf{J} \Delta\mathbf{x} + \Delta\mathbf{x} \mathbf{J}^T \mathbf{J} (\mathbf{N} \Delta\mathbf{x}_h)$$

more detail: <http://www.lingtong.de/2020/04/24/DSO-Null-Space/>
<https://blog.csdn.net/wubaobao1993/article/details/105106301>

4.4. Null Space Effect Elimination

Detailed Derivation of Null Space in DSO

$$\mathbf{E}(\mathbf{x} + \Delta\mathbf{x}) = \mathbf{E}(\mathbf{x} + \Delta\mathbf{x} - \mathbf{N}\Delta\mathbf{x}_h)$$

위 공식의 두 에러함수를 1차 테일러 전개하여 정리하면 다음과 같이 나타낼 수 있다

$$\begin{aligned}\mathbf{E}(\mathbf{x} + \Delta\mathbf{x} - \mathbf{N}\Delta\mathbf{x}_h) &= \mathbf{e}(\mathbf{x} + \Delta\mathbf{x} - \mathbf{N}\Delta\mathbf{x}_h)^T \mathbf{e}(\mathbf{x} + \Delta\mathbf{x} - \mathbf{N}\Delta\mathbf{x}_h) \\ &\simeq (\mathbf{e}(\mathbf{x}) + \mathbf{J}(\Delta\mathbf{x} - \mathbf{N}\Delta\mathbf{x}_h))^T (\mathbf{e}(\mathbf{x}) + \mathbf{J}(\Delta\mathbf{x} - \mathbf{N}\Delta\mathbf{x}_h)) \\ &= \underbrace{\mathbf{e}^T \mathbf{e} + \mathbf{e}^T \mathbf{J} \Delta\mathbf{x} + \Delta\mathbf{x} \mathbf{J}^T \mathbf{e} + \Delta\mathbf{x}^T \mathbf{J}^T \mathbf{J} \Delta\mathbf{x}}_{\mathbf{E}(\mathbf{x} + \Delta\mathbf{x})} - \underbrace{\mathbf{e}^T \mathbf{J} \mathbf{N} \Delta\mathbf{x}_h - (\mathbf{N} \Delta\mathbf{x}_h)^T \mathbf{J}^T \mathbf{e} - (\mathbf{N} \Delta\mathbf{x}_h)^T \mathbf{J}^T \mathbf{J} \Delta\mathbf{x} - \Delta\mathbf{x} \mathbf{J}^T \mathbf{J} (\mathbf{N} \Delta\mathbf{x}_h) + (\mathbf{N} \Delta\mathbf{x}_h)^T \mathbf{J}^T \mathbf{J} (\mathbf{N} \Delta\mathbf{x}_h)}_{\Delta\mathbf{x}_h \text{ part}} \\ &= \mathbf{E}(\mathbf{x} + \Delta\mathbf{x})\end{aligned}$$

따라서 공통부분을 소거한 후 다시 정리하면 다음과 같은 공식이 성립하고

$$(\mathbf{N} \Delta\mathbf{x}_h)^T \mathbf{J}^T \mathbf{J} (\mathbf{N} \Delta\mathbf{x}_h) = \mathbf{e}^T \mathbf{J} \mathbf{N} \Delta\mathbf{x}_h + (\mathbf{N} \Delta\mathbf{x}_h)^T \mathbf{J}^T \mathbf{e} + (\mathbf{N} \Delta\mathbf{x}_h)^T \mathbf{J}^T \mathbf{J} \Delta\mathbf{x} + \Delta\mathbf{x} \mathbf{J}^T \mathbf{J} (\mathbf{N} \Delta\mathbf{x}_h)$$

여기서 Null Space의 선형결합에 해당하는 $\mathbf{N} \Delta\mathbf{x}_h$ 와 에러함수의 기울기를 의미하는 $\mathbf{J}^T \mathbf{e}$ 는 서로 직교하므로 다음이 성립한다.

$$\mathbf{N} \Delta\mathbf{x}_h \mathbf{J}^T \mathbf{e} = 0$$

4.4. Null Space Effect Elimination

Detailed Derivation of Null Space in DSO

$$\mathbf{E}(\mathbf{x} + \Delta\mathbf{x}) = \mathbf{E}(\mathbf{x} + \Delta\mathbf{x} - \mathbf{N}\Delta\mathbf{x}_h)$$

위 공식의 두 에러함수를 1차 테일러 전개하여 정리하면 다음과 같이 나타낼 수 있다

$$\begin{aligned}\mathbf{E}(\mathbf{x} + \Delta\mathbf{x} - \mathbf{N}\Delta\mathbf{x}_h) &= \mathbf{e}(\mathbf{x} + \Delta\mathbf{x} - \mathbf{N}\Delta\mathbf{x}_h)^T \mathbf{e}(\mathbf{x} + \Delta\mathbf{x} - \mathbf{N}\Delta\mathbf{x}_h) \\ &\simeq (\mathbf{e}(\mathbf{x}) + \mathbf{J}(\Delta\mathbf{x} - \mathbf{N}\Delta\mathbf{x}_h))^T (\mathbf{e}(\mathbf{x}) + \mathbf{J}(\Delta\mathbf{x} - \mathbf{N}\Delta\mathbf{x}_h)) \\ &= \underbrace{\mathbf{e}^T \mathbf{e} + \mathbf{e}^T \mathbf{J} \Delta\mathbf{x} + \Delta\mathbf{x} \mathbf{J}^T \mathbf{e} + \Delta\mathbf{x}^T \mathbf{J}^T \mathbf{J} \Delta\mathbf{x}}_{\mathbf{E}(\mathbf{x} + \Delta\mathbf{x})} - \underbrace{\mathbf{e}^T \mathbf{J} \mathbf{N} \Delta\mathbf{x}_h - (\mathbf{N} \Delta\mathbf{x}_h)^T \mathbf{J}^T \mathbf{e} - (\mathbf{N} \Delta\mathbf{x}_h)^T \mathbf{J}^T \mathbf{J} \Delta\mathbf{x} - \Delta\mathbf{x} \mathbf{J}^T \mathbf{J} (\mathbf{N} \Delta\mathbf{x}_h) + (\mathbf{N} \Delta\mathbf{x}_h)^T \mathbf{J}^T \mathbf{J} (\mathbf{N} \Delta\mathbf{x}_h)}_{\Delta\mathbf{x}_h \text{ part}} \\ &= \mathbf{E}(\mathbf{x} + \Delta\mathbf{x})\end{aligned}$$

따라서 공통부분을 소거한 후 다시 정리하면 다음과 같은 공식이 성립하고

$$(\mathbf{N} \Delta\mathbf{x}_h)^T \mathbf{J}^T \mathbf{J} (\mathbf{N} \Delta\mathbf{x}_h) = \mathbf{e}^T \mathbf{J} \mathbf{N} \Delta\mathbf{x}_h + (\mathbf{N} \Delta\mathbf{x}_h)^T \mathbf{J}^T \mathbf{e} + (\mathbf{N} \Delta\mathbf{x}_h)^T \mathbf{J}^T \mathbf{J} \Delta\mathbf{x} + \Delta\mathbf{x} \mathbf{J}^T \mathbf{J} (\mathbf{N} \Delta\mathbf{x}_h)$$

여기서 Null Space의 선형결합에 해당하는 $\mathbf{N} \Delta\mathbf{x}_h$ 와 에러함수의 기울기를 의미하는 $\mathbf{J}^T \mathbf{e}$ 는 서로 직교하므로 다음이 성립한다.

$$\mathbf{N} \Delta\mathbf{x}_h \mathbf{J}^T \mathbf{e} = 0$$

따라서 남은 식들을 정리하면 다음 공식이 성립한다

$$\mathbf{N} \Delta\mathbf{x}_h = \Delta\mathbf{x}$$

more detail: <http://www.lingtong.de/2020/04/24/DSO-Null-Space/>
<https://blog.csdn.net/wubaobao1993/article/details/105106301>

4.4. Null Space Effect Elimination

Detailed Derivation of Null Space in DSO

$$\mathbf{N}\Delta\mathbf{x}_h = \Delta\mathbf{x}$$

하지만 위 공식은 실제로 서로 일치하지 않으므로 이를 $\Delta\mathbf{x}_h$ 에 대한 Least Square를 활용하여 근사해를 구하면 $\|\mathbf{N}\Delta\mathbf{x}_h - \Delta\mathbf{x}\|$ 의 크기가 가장 최소가 되는 근사해를 구할 수 있다.

more detail: <http://www.lingtonq.de/2020/04/24/DSO-Null-Space/>
<https://blog.csdn.net/wubaobao1993/article/details/105106301>

4.4. Null Space Effect Elimination

Detailed Derivation of Null Space in DSO

$$\mathbf{N}\Delta\mathbf{x}_h = \Delta\mathbf{x}$$

하지만 위 공식은 실제로 서로 일치하지 않으므로 이를 $\Delta\mathbf{x}_h$ 에 대한 Least Square를 활용하여 근사해를 구하면 $\|\mathbf{N}\Delta\mathbf{x}_h - \Delta\mathbf{x}\|$ 의 크기가 가장 최소가 되는 근사해를 구할 수 있다.

이 때 $\mathbf{N}^T\mathbf{N}$ 의 역행렬이 존재하는가 아닌가에 따라 다음과 같이 해를 구할 수 있다

$$\Delta\mathbf{x}_h = \mathbf{N}^\dagger \Delta\mathbf{x} \quad \text{if, } \mathbf{N}\mathbf{N}^T \text{ is invertible.}$$

$$\Delta\mathbf{x}_h = \mathbf{V}\mathbf{D}^\dagger \mathbf{U}^T \Delta\mathbf{x} \quad \text{if, } \mathbf{N}\mathbf{N}^T \text{ is not invertible.}$$

more detail: <http://www.lingtonq.de/2020/04/24/DSO-Null-Space/>
<https://blog.csdn.net/wubaobao1993/article/details/105106301>

4.4. Null Space Effect Elimination

Detailed Derivation of Null Space in DSO

$$\mathbf{N}\Delta\mathbf{x}_h = \Delta\mathbf{x}$$

하지만 위 공식은 실제로 서로 일치하지 않으므로 이를 $\Delta\mathbf{x}_h$ 에 대한 Least Square를 활용하여 근사해를 구하면 $\|\mathbf{N}\Delta\mathbf{x}_h - \Delta\mathbf{x}\|$ 의 크기가 가장 최소가 되는 근사해를 구할 수 있다.

이 때 $\mathbf{N}^T\mathbf{N}$ 의 역행렬이 존재하는가 아닌가에 따라 다음과 같이 해를 구할 수 있다

$$\begin{aligned}\Delta\mathbf{x}_h &= \mathbf{N}^\dagger \Delta\mathbf{x} && \text{if, } \mathbf{N}\mathbf{N}^T \text{ is invertible.} \\ \Delta\mathbf{x}_h &= \mathbf{V}\mathbf{D}^\dagger \mathbf{U}^T \Delta\mathbf{x} && \text{if, } \mathbf{N}\mathbf{N}^T \text{ is not invertible.}\end{aligned}$$

결론적으로 다음과 같이 최적의 업데이트 값 $\Delta\mathbf{x}^*$ 를 계산할 수 있다

$$\Delta\mathbf{x}^* = \Delta\mathbf{x} - \mathbf{N}\Delta\mathbf{x}_h \Rightarrow \Delta\mathbf{x}^* = \Delta\mathbf{x} - \mathbf{N}\mathbf{N}^\dagger \Delta\mathbf{x}$$

more detail: <http://www.lingtonq.de/2020/04/24/DSO-Null-Space/>
<https://blog.csdn.net/wubaobao1993/article/details/105106301>

4.4. Null Space Effect Elimination

Detailed Derivation of Null Space in DSO

Null Space의 영향이 제거된 \mathbf{H}^* , \mathbf{b}^* 는 다음과 같이 유도할 수 있다

more detail: <http://www.lingtonq.de/2020/04/24/DSO-Null-Space/>
<https://blog.csdn.net/wubaobao1993/article/details/105106301>

4.4. Null Space Effect Elimination

Detailed Derivation of Null Space in DSO

Null Space의 영향이 제거된 \mathbf{H}^* , \mathbf{b}^* 는 다음과 같이 유도할 수 있다

$$\begin{aligned}\Delta \mathbf{x}^T \mathbf{b}^* &= \Delta \mathbf{x}^T \mathbf{b} - \Delta \mathbf{x}_h^T \mathbf{b}_n \\ &= \Delta \mathbf{x}^T \mathbf{b} - (\mathbf{N}^\dagger \Delta \mathbf{x})^T \mathbf{N}^T \mathbf{J}^T \mathbf{e} \\ &= \Delta \mathbf{x}^T \mathbf{b} - \Delta \mathbf{x}^T (\mathbf{N} \mathbf{N}^\dagger)^T \mathbf{b} \\ &= \text{where, } \mathbf{b}_n = \mathbf{J}_n \mathbf{e} \quad \mathbf{J}_n = \frac{\partial \mathbf{e}}{\partial \Delta \mathbf{x}_h} = \frac{\partial \mathbf{e}}{\partial \Delta \mathbf{x}} \frac{\partial \Delta \mathbf{x}}{\partial \Delta \mathbf{x}_h} = \mathbf{J} \mathbf{N}\end{aligned}$$

$$\begin{aligned}\Delta \mathbf{x}^T \mathbf{H}^* \Delta \mathbf{x} &= \Delta \mathbf{x}^T \mathbf{H} \Delta \mathbf{x} - \Delta \mathbf{x}_h^T \mathbf{H}_n \Delta \mathbf{x}_h \\ &= \Delta \mathbf{x}^T \mathbf{H} \Delta \mathbf{x} - (\mathbf{N}^\dagger \Delta \mathbf{x})^T \mathbf{N}^T \mathbf{J}^T \mathbf{J} \mathbf{N} (\mathbf{N}^\dagger \Delta \mathbf{x}) \\ &= \Delta \mathbf{x}^T \mathbf{H} \Delta \mathbf{x} - \Delta \mathbf{x}^T (\mathbf{N} \mathbf{N}^\dagger)^T \mathbf{H} (\mathbf{N} \mathbf{N}^\dagger) \Delta \mathbf{x} \\ &\text{where, } \mathbf{H}_n = \mathbf{J}_n^T \mathbf{J}_n, \quad \mathbf{J}_n = \frac{\partial \mathbf{e}}{\partial \Delta \mathbf{x}_h} = \frac{\partial \mathbf{e}}{\partial \Delta \mathbf{x}} \frac{\partial \Delta \mathbf{x}}{\partial \Delta \mathbf{x}_h} = \mathbf{J} \mathbf{N}.\end{aligned}$$

more detail: <http://www.lingtonq.de/2020/04/24/DSO-Null-Space/>
<https://blog.csdn.net/wubaobao1993/article/details/105106301>

4.4. Null Space Effect Elimination

Detailed Derivation of Null Space in DSO

Null Space의 영향이 제거된 \mathbf{H}^* , \mathbf{b}^* 는 다음과 같이 유도할 수 있다

$$\begin{aligned}\Delta \mathbf{x}^T \mathbf{b}^* &= \Delta \mathbf{x}^T \mathbf{b} - \Delta \mathbf{x}_h^T \mathbf{b}_n \\ &= \Delta \mathbf{x}^T \mathbf{b} - (\mathbf{N}^\dagger \Delta \mathbf{x})^T \mathbf{N}^T \mathbf{J}^T \mathbf{e} \\ &= \Delta \mathbf{x}^T \mathbf{b} - \Delta \mathbf{x}^T (\mathbf{N} \mathbf{N}^\dagger)^T \mathbf{b} \\ &= \text{where, } \mathbf{b}_n = \mathbf{J}_n \mathbf{e} \quad \mathbf{J}_n = \frac{\partial \mathbf{e}}{\partial \Delta \mathbf{x}_h} = \frac{\partial \mathbf{e}}{\partial \Delta \mathbf{x}} \frac{\partial \Delta \mathbf{x}}{\partial \Delta \mathbf{x}_h} = \mathbf{J} \mathbf{N}\end{aligned}$$

$$\begin{aligned}\Delta \mathbf{x}^T \mathbf{H}^* \Delta \mathbf{x} &= \Delta \mathbf{x}^T \mathbf{H} \Delta \mathbf{x} - \Delta \mathbf{x}_h^T \mathbf{H}_n \Delta \mathbf{x}_h \\ &= \Delta \mathbf{x}^T \mathbf{H} \Delta \mathbf{x} - (\mathbf{N}^\dagger \Delta \mathbf{x})^T \mathbf{N}^T \mathbf{J}^T \mathbf{J} \mathbf{N} (\mathbf{N}^\dagger \Delta \mathbf{x}) \\ &= \Delta \mathbf{x}^T \mathbf{H} \Delta \mathbf{x} - \Delta \mathbf{x}^T (\mathbf{N} \mathbf{N}^\dagger)^T \mathbf{H} (\mathbf{N} \mathbf{N}^\dagger) \Delta \mathbf{x} \\ &\text{where, } \mathbf{H}_n = \mathbf{J}_n^T \mathbf{J}_n, \quad \mathbf{J}_n = \frac{\partial \mathbf{e}}{\partial \Delta \mathbf{x}_h} = \frac{\partial \mathbf{e}}{\partial \Delta \mathbf{x}} \frac{\partial \Delta \mathbf{x}}{\partial \Delta \mathbf{x}_h} = \mathbf{J} \mathbf{N}.\end{aligned}$$

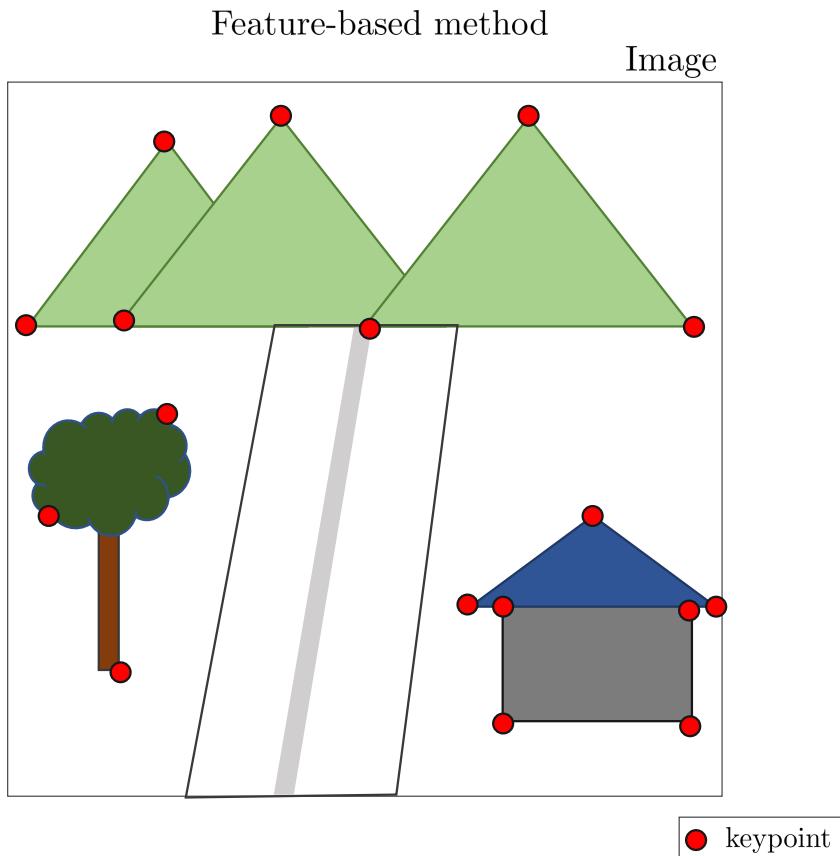
따라서 최종적으로 다음의 연산을 통해 Null Space의 영향을 제거할 수 있다

$$\begin{aligned}\mathbf{b}^* &= \mathbf{b} - (\mathbf{N} \mathbf{N}^\dagger)^T \mathbf{b} \\ \mathbf{H}^* &= \mathbf{H} - (\mathbf{N} \mathbf{N}^\dagger)^T \mathbf{H} (\mathbf{N} \mathbf{N}^\dagger)\end{aligned}$$

more detail: <http://www.lingtonq.de/2020/04/24/DSO-Null-Space/>
<https://blog.csdn.net/wubaobao1993/article/details/105106301>

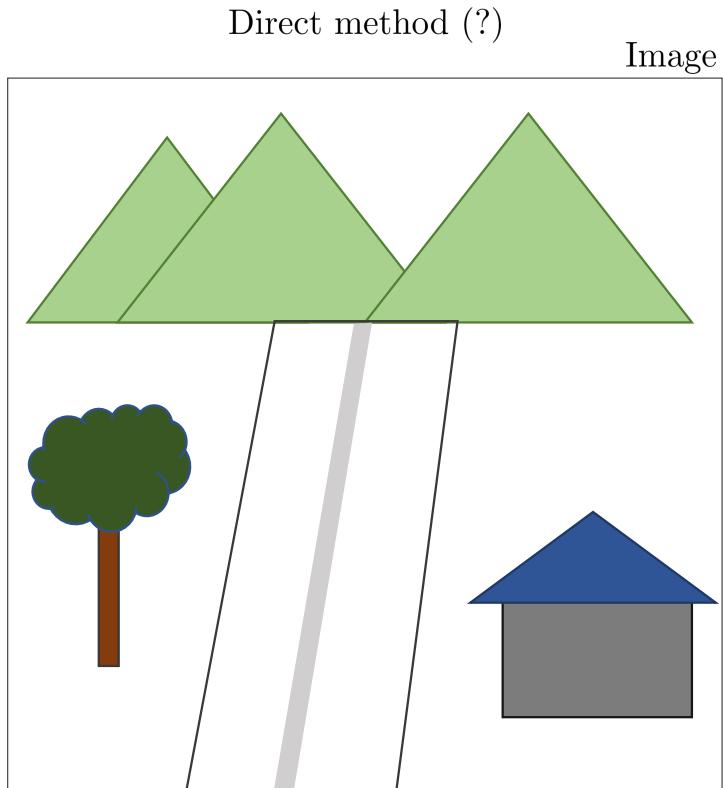
5. Pixel Selection

5.1. Pixel Selection → Make Histogram



ORB-SLAM2와 같은 feature-based method의 경우 이미지 내에서 특정 포인트를 추출할 때 feature extraction 알고리즘을 사용하여 키포인트를 추출한다. 그 다음 해당 키포인트를 3차원 점으로 Unprojection $\pi^{-1}(\cdot)$ 하고 다음 프레임에서 다시 2차원 점으로 Reprojection $\pi(\cdot)$ 해서 Reprojection Error를 사용하여 카메라 모션을 계산한다.

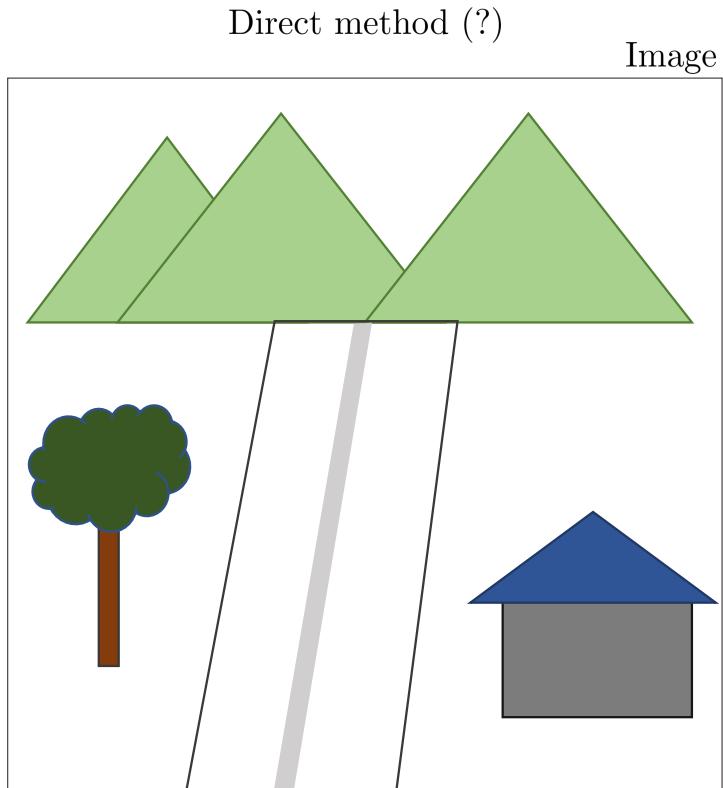
5.1. Pixel Selection → Make Histogram



ORB-SLAM2와 같은 feature-based method의 경우 이미지 내에서 특정 포인트를 추출할 때 feature extraction 알고리즘을 사용하여 키포인트를 추출한다. 그 다음 해당 키포인트를 3차원 점으로 Unprojection $\pi^{-1}(\cdot)$ 하고 다음 프레임에서 다시 2차원 점으로 Reprojection $\pi(\cdot)$ 해서 Reprojection Error를 사용하여 카메라 모션을 계산한다.

하지만 direct method를 사용하는 DSO의 경우 feature를 별도로 추출하는 과정없이 특정 픽셀의 밝기 오차를 계산하기 때문에 특정 포인트를 측정하는 방법 또한 feature-based와 달라지게 된다.

5.1. Pixel Selection → Make Histogram



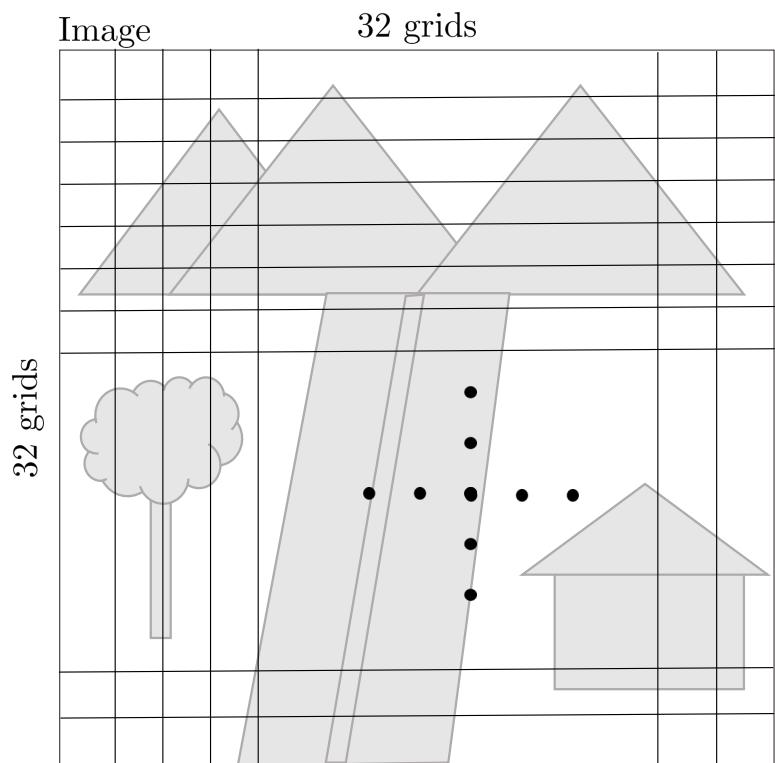
ORB-SLAM2와 같은 feature-based method의 경우 이미지 내에서 특정 포인트를 추출할 때 feature extraction 알고리즘을 사용하여 키포인트를 추출한다. 그 다음 해당 키포인트를 3차원 점으로 Unprojection $\pi^{-1}(\cdot)$ 하고 다음 프레임에서 다시 2차원 점으로 Reprojection $\pi(\cdot)$ 해서 Reprojection Error를 사용하여 카메라 모션을 계산한다.

하지만 direct method를 사용하는 DSO의 경우 feature를 별도로 추출하는 과정없이 특정 픽셀의 밝기 오차를 계산하기 때문에 특정 포인트를 측정하는 방법 또한 feature-based와 달라지게 된다.

DSO에서는 이미지에서 특정 포인트를 추출하기 위해 **Gradient Histogram** 방법과 **Dynamic Grid** 방법을 사용했다. 해당 방법은 DSO 논문에서도 자세한 설명이 부족하고 참고 논문이 없으며 구글링해도 나오는 정보가 거의 없으므로 해당 섹션에서 다루는 내용이 100% 정확하다고 볼 수 없다. 해당 내용은 논문을 볼 때 참고용으로 보면 좋을 것 같다.

참조 [github](https://github.com/alalagong/DSO)
<https://github.com/alalagong/DSO>

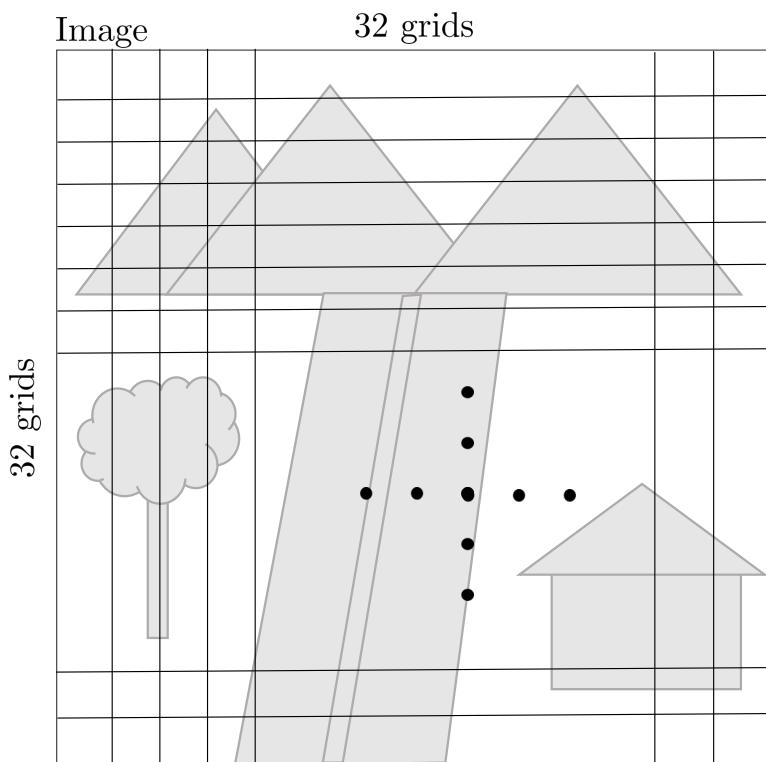
5.1. Pixel Selection ➔ Make Histogram



Make Histogram

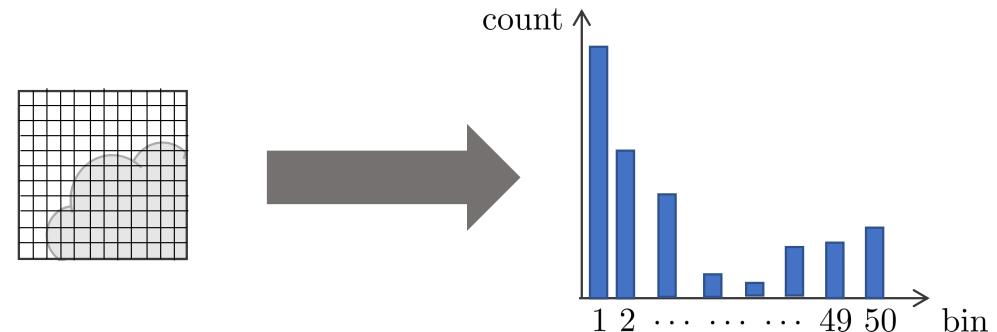
- 1) 우선 새로운 이미지가 들어오면 Gradient Histogram을 생성하기 위해 이미지를 32x32 영역으로 나눈다. (grayscale image)

5.1. Pixel Selection ➔ Make Histogram

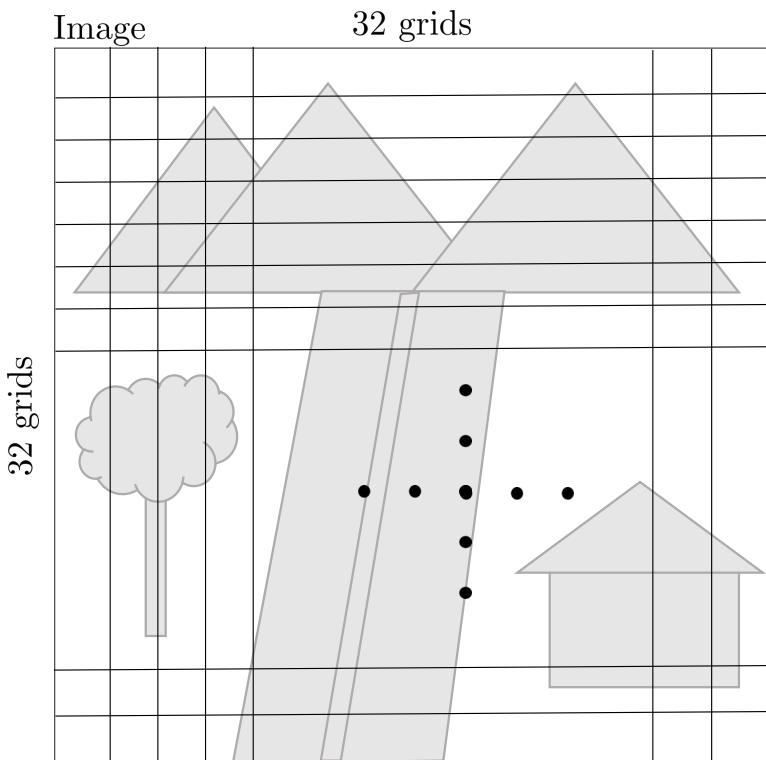


Make Histogram

- 1) 우선 새로운 이미지가 들어오면 Gradient Histogram을 생성하기 위해 이미지를 32x32 영역으로 나눈다. (grayscale image)
- 2) 하나의 영역을 확대해서 보면 아래 그림과 같이 여러 픽셀들로 구성되어 있고 각 픽셀마다 고유의 밝기(intensity) 값을 가지고 있다. 이를 이용해 Image Gradient(dx, dy)를 구할 수 있고 이를 이용해 해당 영역에 대한 히스토그램을 만들 수 있다.

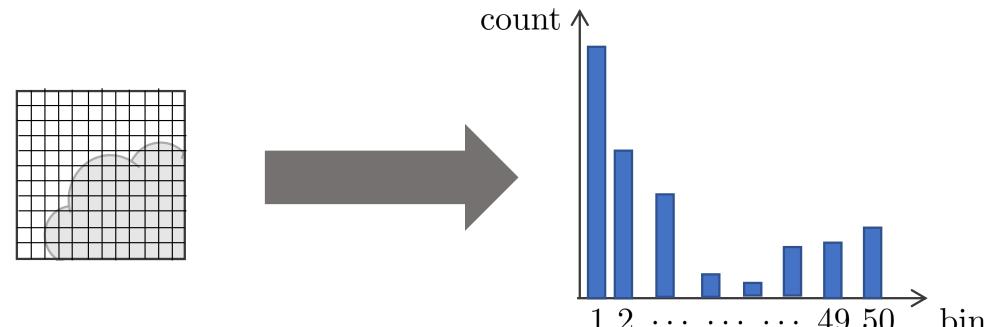


5.1. Pixel Selection → Make Histogram



Make Histogram

- 1) 우선 새로운 이미지가 들어오면 Gradient Histogram을 생성하기 위해 이미지를 32x32 영역으로 나눈다. (grayscale image)
- 2) 하나의 영역을 확대해서 보면 아래 그림과 같이 여러 픽셀들로 구성되어 있고 각 픽셀마다 고유의 밝기(intensity) 값을 가지고 있다. 이를 이용해 Image Gradient(dx, dy)를 구할 수 있고 이를 이용해 해당 영역에 대한 히스토그램을 만들 수 있다.

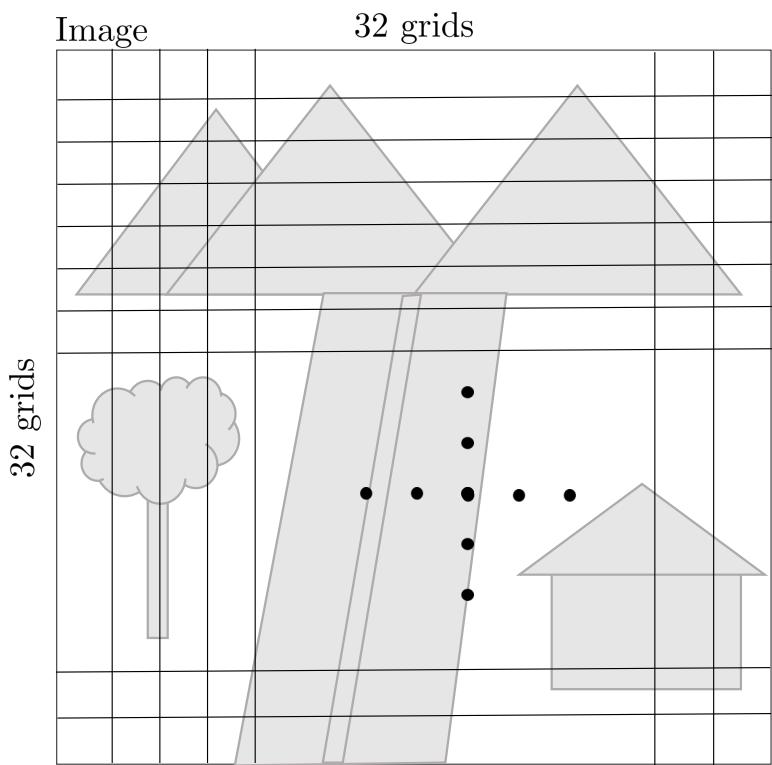


- 3) 이 때 Gradient dx, dy 는 원래 0~255 값을 가지지만 DSO에서는 한 픽셀의 $bin = \sqrt{dx^2 + dy^2}$ 값을 사용하여 히스토그램을 생성했다.

$$0 < \sqrt{dx^2 + dy^2} < 360$$

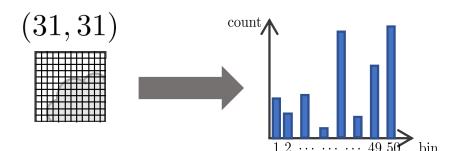
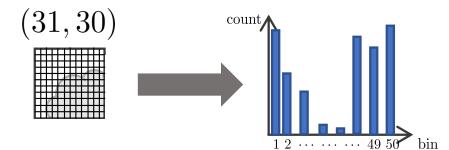
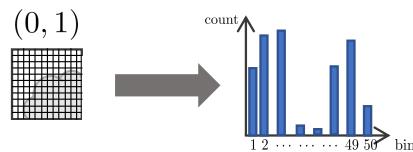
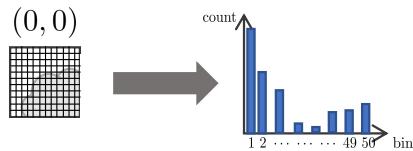
bin은 위와 같은 범위를 가지지만 이를 1~50 범위로 제한하여 50이 넘는 경우 50으로 설정하여 히스토그램의 범위를 단순화했다.

5.1. Pixel Selection → Make Histogram

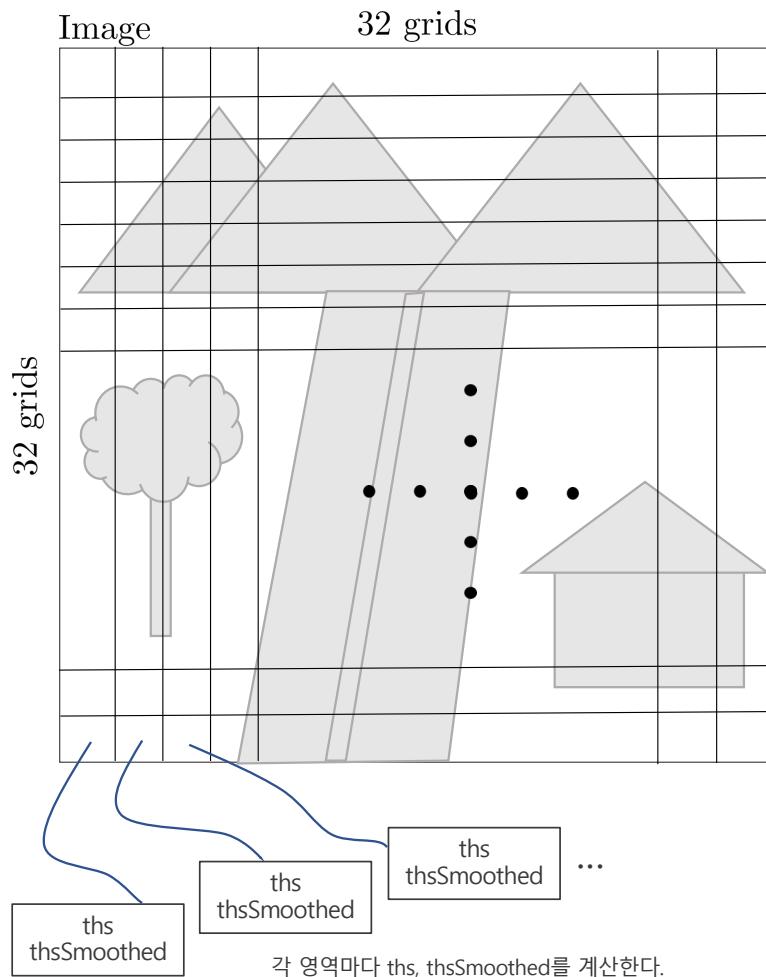


Make Histogram

4) 이러한 히스토그램을 32×32 모든 영역에 대해 각각 계산한다.

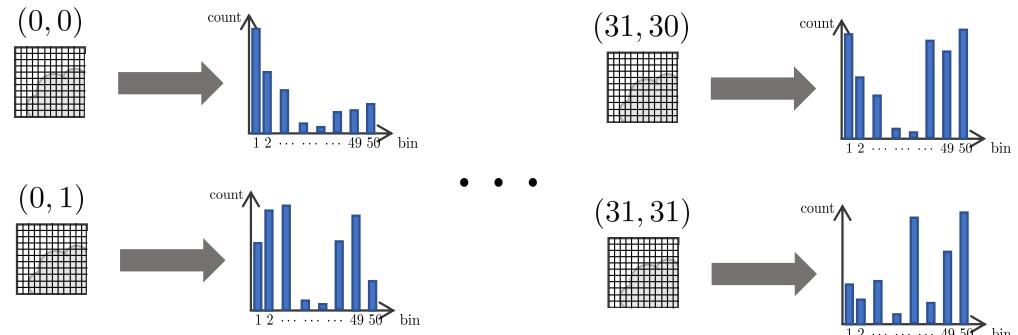


5.1. Pixel Selection → Make Histogram

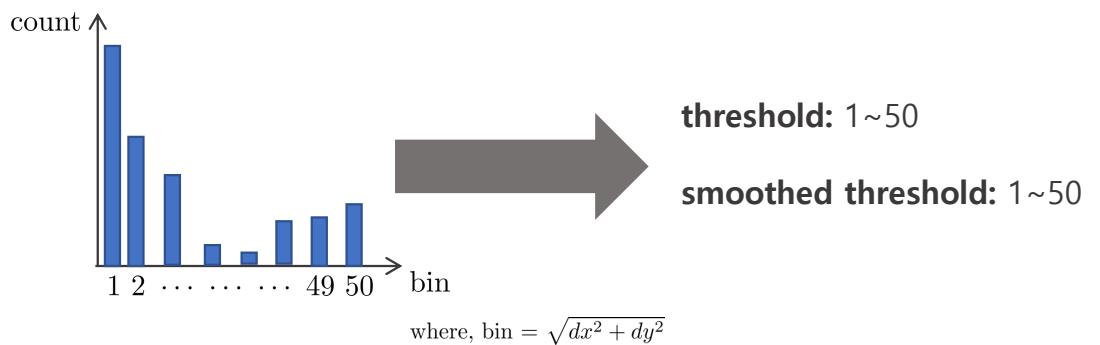


Make Histogram

4) 이러한 히스토그램을 32x32 모든 영역에 대해 각각 계산한다.

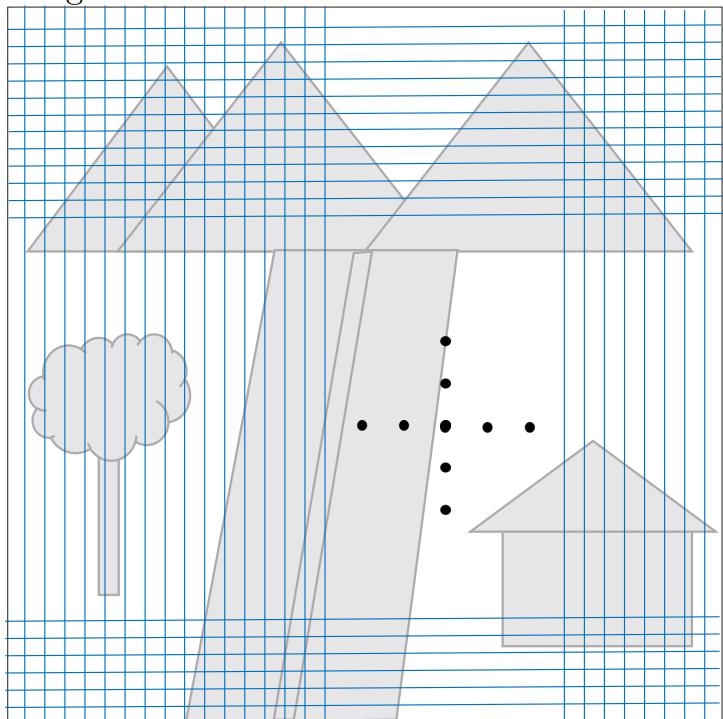


5) 히스토그램에서 Total count의 50%가 되는 bin 값을 1-50 순서대로 찾는다.
예를들어 히스토그램의 모든 bin에서 총 1000개가 카운트 되었다면 1번 bin부터 순서대로 카운트 값을 빼면서 해당 값이 500보다 작아지는 bin을 threshold로 설정한다.
또한 주변 3x3 영역의 threshold의 평균을 사용한 smoothed threshold 값 또한 설정한다.



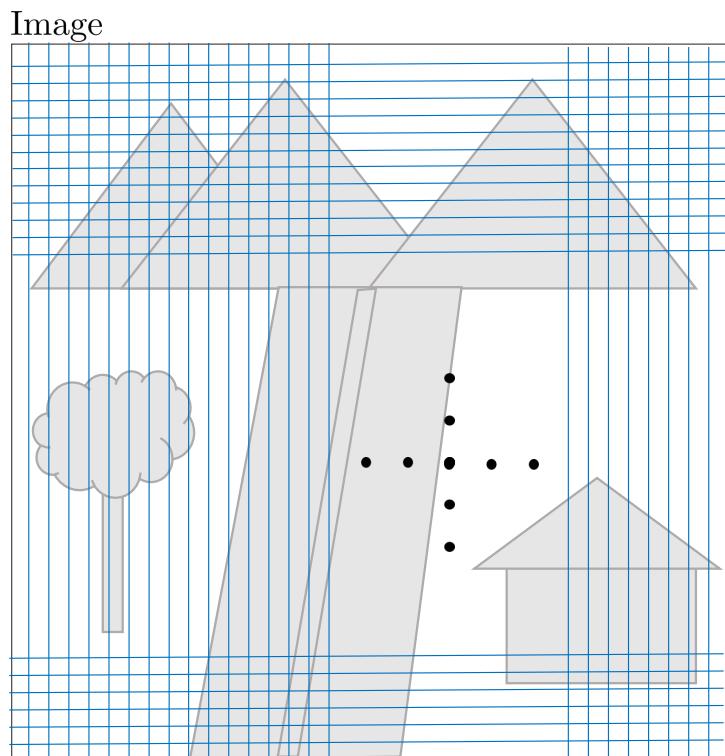
5.2. Pixel Selection → Dynamic Grid

Image



한 이미지에 대한 32×32 히스토그램을 생성하고 threshold 값을 설정하였으면 다음으로 Dynamic Grid 방법을 통해 이미지 내에서 픽셀을 선택한다. 이를 통해 High gradient 영역에 대부분의 포인트가 추출되는 것을 방지하고 한 이미지 내에서 추출할 수 있는 총 포인트의 개수($N=2000$)를 관리할 수 있는 장점을 지닌다.

5.2. Pixel Selection → Dynamic Grid



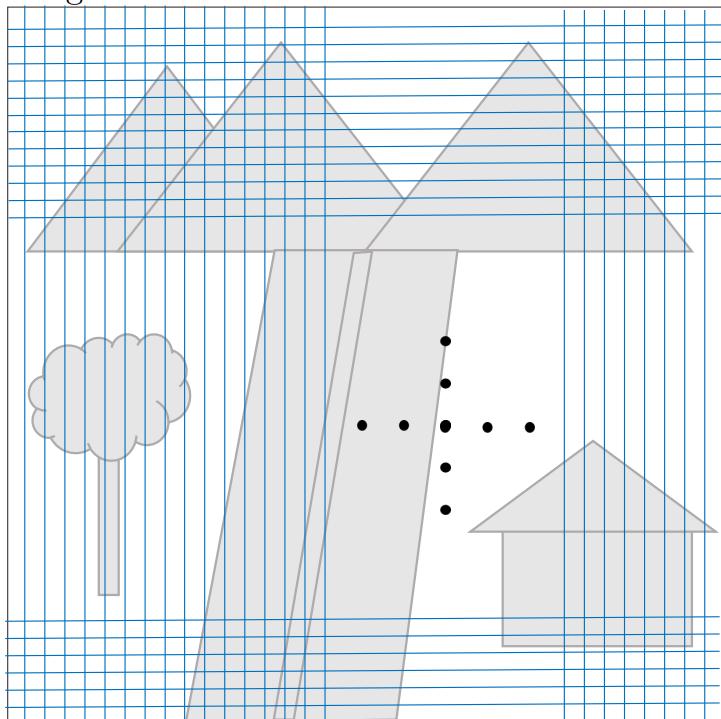
한 이미지에 대한 32×32 히스토그램을 생성하고 threshold 값을 설정하였으면 다음으로 Dynamic Grid 방법을 통해 이미지 내에서 픽셀을 선택한다. 이를 통해 High gradient 영역에 대부분의 포인트가 추출되는 것을 방지하고 한 이미지 내에서 추출할 수 있는 총 포인트의 개수($N=2000$)를 관리할 수 있는 장점을 지닌다.

Dynamic Grid

- 1) 하나의 Grid 영역의 크기를 설정한다. DSO에서는 초기값으로 12×12 [pixel]를 사용했다. 640×480 이미지 기준으로 봤을 때 일반적으로 32×32 histogram 영역보다 작은 크기를 사용한다. Grid 크기를 설정했으면 이미지 전체를 Grid로 나눈다.

5.2. Pixel Selection → Dynamic Grid

Image

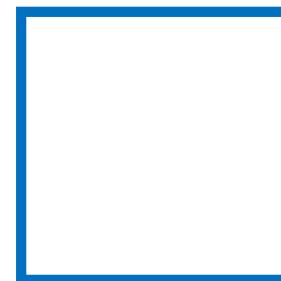


한 이미지에 대한 32×32 히스토그램을 생성하고 threshold 값을 설정하였으면 다음으로 Dynamic Grid 방법을 통해 이미지 내에서 픽셀을 선택한다. 이를 통해 High gradient 영역에 대부분의 포인트가 추출되는 것을 방지하고 한 이미지 내에서 추출할 수 있는 총 포인트의 개수($N=2000$)를 관리할 수 있는 장점을 지닌다.

Dynamic Grid

1) 하나의 Grid 영역의 크기를 설정한다. DSO에서는 초기값으로 12×12 [pixel]를 사용했다. 640×480 이미지 기준으로 봤을 때 일반적으로 32×32 histogram 영역보다 작은 크기를 사용한다. Grid 크기를 설정했으면 이미지 전체를 Grid로 나눈다.

2) 한 Grid 영역을 확대해서 보면 아래와 같다.

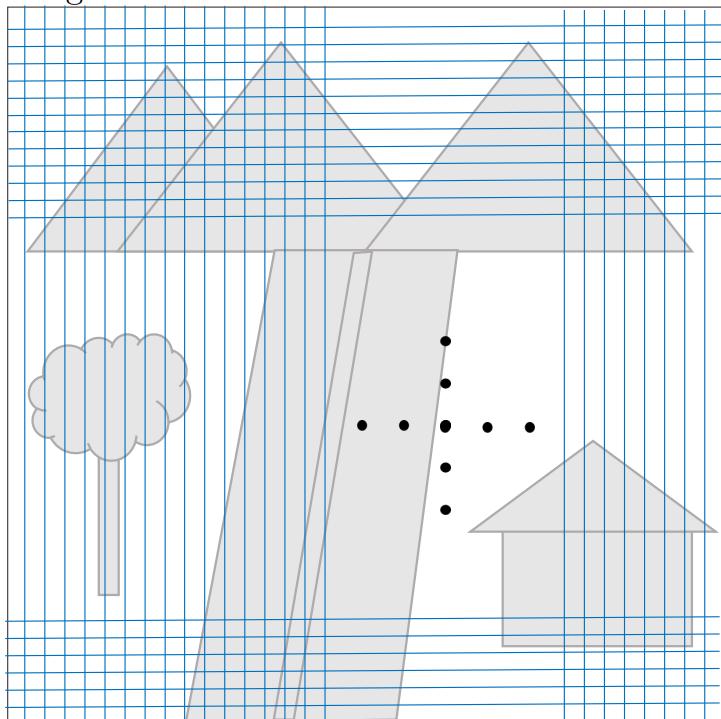


참조 github

<https://github.com/alalagong/DSO>

5.2. Pixel Selection → Dynamic Grid

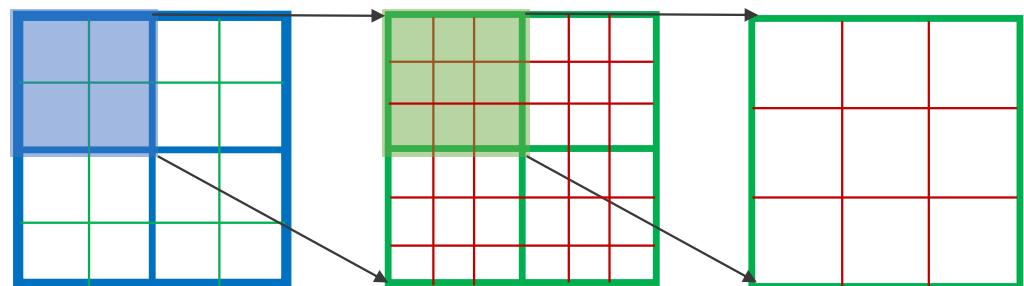
Image



한 이미지에 대한 32×32 히스토그램을 생성하고 threshold 값을 설정하였으면 다음으로 Dynamic Grid 방법을 통해 이미지 내에서 픽셀을 선택한다. 이를 통해 High gradient 영역에 대부분의 포인트가 추출되는 것을 방지하고 한 이미지 내에서 추출할 수 있는 총 포인트의 개수($N=2000$)를 관리할 수 있는 장점을 지닌다.

Dynamic Grid

- 1) 하나의 Grid 영역의 크기를 설정한다. DSO에서는 초기값으로 12×12 [pixel]를 사용했다. 640×480 이미지 기준으로 봤을 때 일반적으로 32×32 histogram 영역보다 작은 크기를 사용한다. Grid 크기를 설정했으면 이미지 전체를 Grid로 나눈다.
- 2) 한 Grid 영역을 확대해서 보면 아래와 같다. 이 때 한 영역에서 총 3개의 피라미드 이미지의 Gradient를 사용한다. Alalagong님의 자료를 참조하여 피라미드 별로 파란색(Lv2), 초록색(Lv1), 빨간색(Lv0, 원본)로 표시하면 아래 그림과 같고 순서대로 해당 영역의 픽셀을 탐색한다.

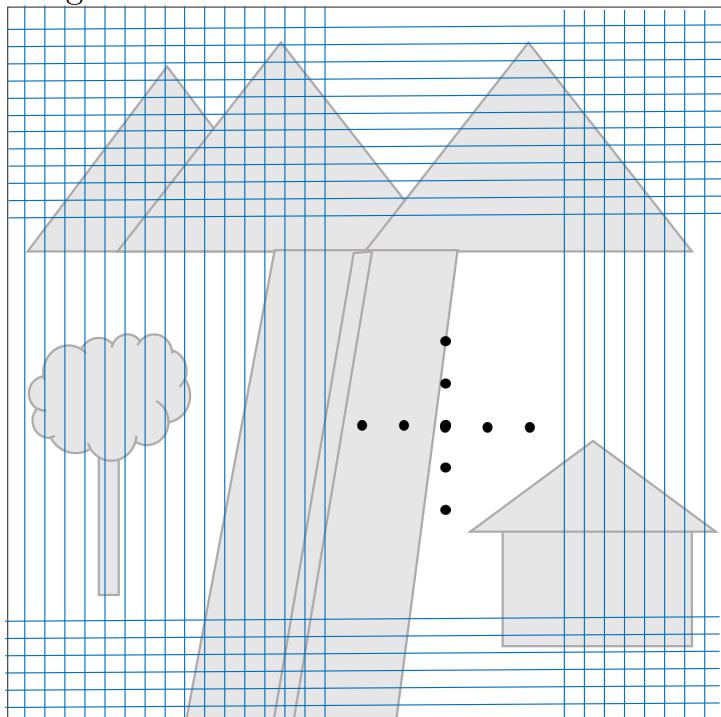


참조 [github](#)

<https://github.com/alalagong/DSO>

5.2. Pixel Selection → Dynamic Grid

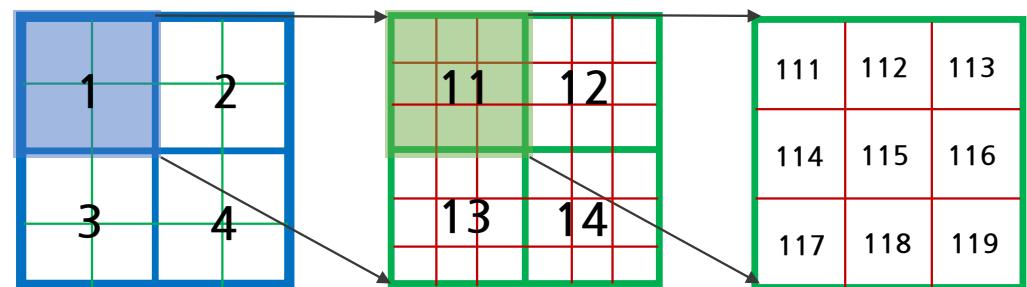
Image



한 이미지에 대한 32×32 히스토그램을 생성하고 threshold 값을 설정하였으면 다음으로 Dynamic Grid 방법을 통해 이미지 내에서 픽셀을 선택한다. 이를 통해 High gradient 영역에 대부분의 포인트가 추출되는 것을 방지하고 한 이미지 내에서 추출할 수 있는 총 포인트의 개수($N=2000$)를 관리할 수 있는 장점을 지닌다.

Dynamic Grid

- 1) 하나의 Grid 영역의 크기를 설정한다. DSO에서는 초기값으로 12×12 [pixel]를 사용했다. 640×480 이미지 기준으로 봤을 때 일반적으로 32×32 histogram 영역보다 작은 크기를 사용한다. Grid 크기를 설정했으면 이미지 전체를 Grid로 나눈다.
- 2) 한 Grid 영역을 확대해서 보면 아래와 같다. 이 때 한 영역에서 총 3개의 피라미드 이미지의 Gradient를 사용한다. Alalagong님의 자료를 참조하여 피라미드 별로 파란색(Lv2), 초록색(Lv1), 빨간색(Lv0, 원본)로 표시하면 아래 그림과 같고 순서대로 해당 영역의 픽셀을 탐색한다.

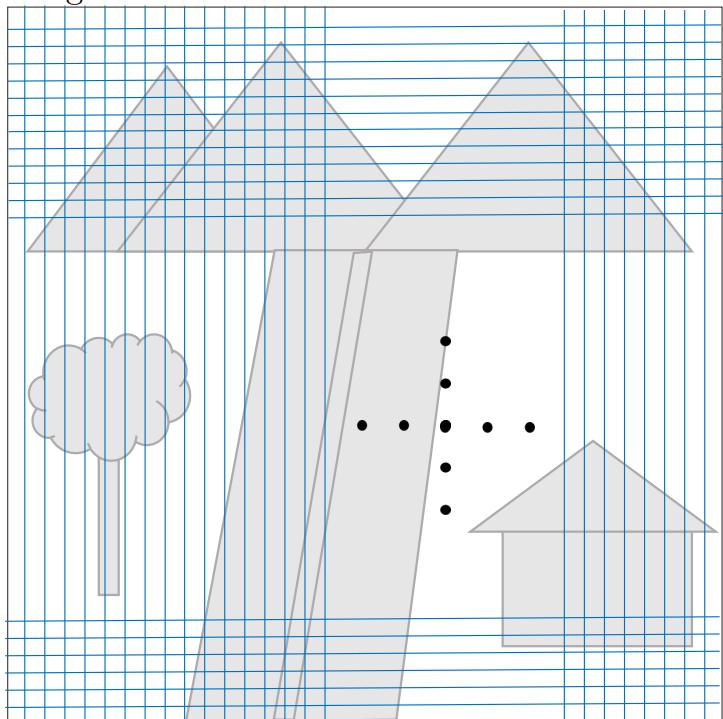


참조 [github](#)

<https://github.com/alalagong/DSO>

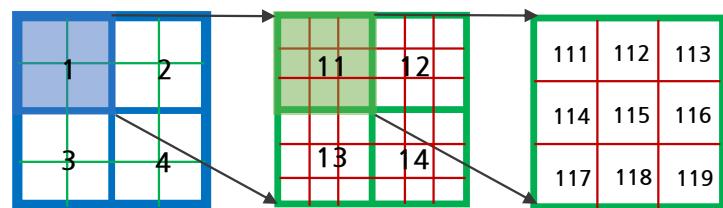
5.2. Pixel Selection → Dynamic Grid

Image

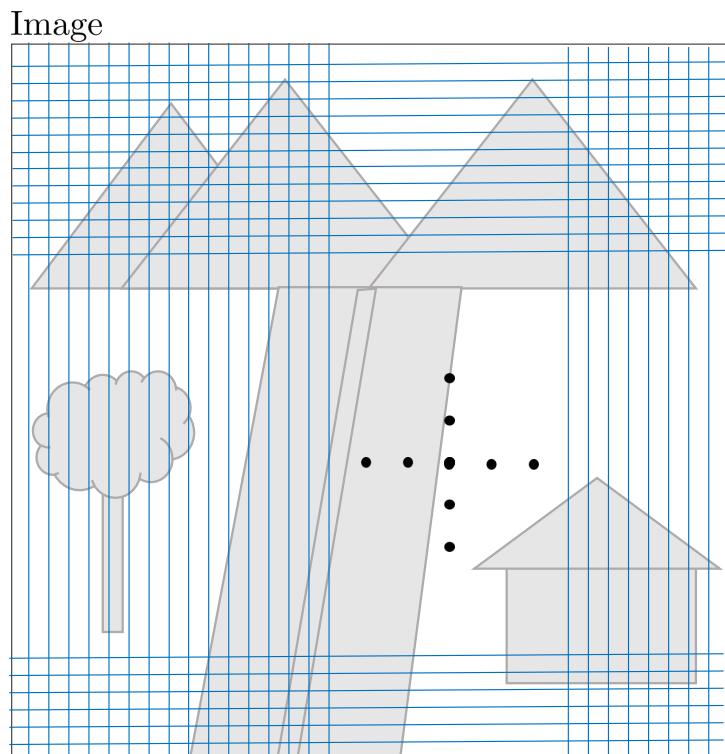


Dynamic Grid

3) 피라미드 레벨이 낮은 곳(Lv0,원본)부터 한 픽셀 씩 루프를 돌면서 해당 영역 히스토그램의 Smoothed threshold 값과 픽셀의 Squared Gradient $\sqrt{dx^2 + dy^2}$ 값을 비교한다.
(111→112→113→…→119→11→1→121→122→123→…→129→12→2→…)

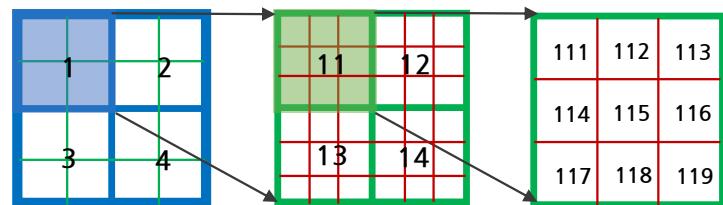


5.2. Pixel Selection → Dynamic Grid



Dynamic Grid

3) 피라미드 레벨이 낮은 곳(Lv0,원본)부터 한 픽셀 씩 루프를 돌면서 해당 영역 히스토그램의 Smoothed threshold 값과 픽셀의 Squared Gradient $\sqrt{dx^2 + dy^2}$ 값을 비교한다.
(111→112→113→…→119→11→1→121→122→123→…→129→12→2→…)

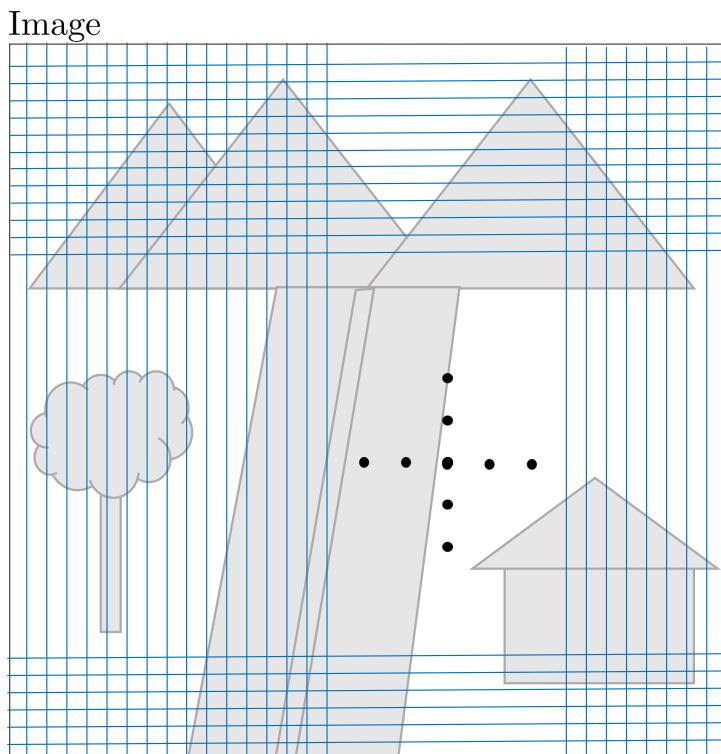


이 때, 특정 픽셀 값이 Smoothed threshold보다 크면 다음 단계로써 해당 픽셀의 (dx, dy) 값과 랜덤한 방향벡터인 randdir과 norm을 계산한 후 해당 값이 0보다 큰지 검사한다.

$$\text{randdir} \cdot (dx, dy) > 0$$

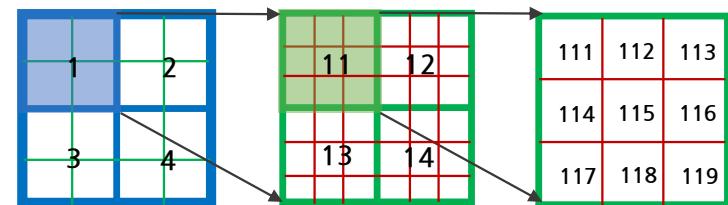
최종적으로 norm까지 특정값보다 크다면 해당 픽셀을 를 대표하는 픽셀로 선정하고 루프를 탈출한다.

5.2. Pixel Selection → Dynamic Grid



Dynamic Grid

3) 피라미드 레벨이 낮은 곳(Lv0,원본)부터 한 픽셀 씩 루프를 돌면서 해당 영역 히스토그램의 Smoothed threshold 값과 픽셀의 Squared Gradient $\sqrt{dx^2 + dy^2}$ 값을 비교한다.
(111→112→113→…→119→11→1→121→122→123→…→129→12→2→…)



이 때, 특정 픽셀 값이 Smoothed threshold보다 크면 다음 단계로써 해당 픽셀의 (dx, dy) 값과 랜덤한 방향벡터인 randdir과 norm을 계산한 후 해당 값이 0보다 큰지 검사한다.

$$\text{randdir} \cdot (dx, dy) > 0$$

최종적으로 norm까지 특정값보다 크다면 해당 픽셀을 ■를 대표하는 픽셀로 선정하고 루프를 탈출한다.

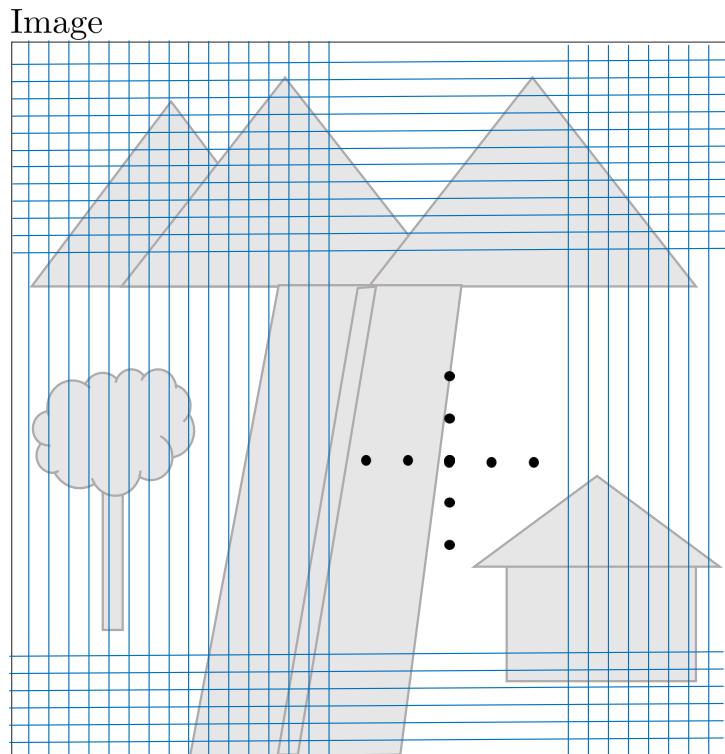
ex) 만약 115 픽셀의 $\sqrt{dx^2 + dy^2}$ 값이 Smoothed threshold보다 큰 경우

$$\text{randdir} \cdot (dx_{115}, dy_{115}) > 0$$

를 검사하고 이 또한 만족하는 경우 115 픽셀을 ■ 영역의 대표 픽셀로 선정한다.

랜덤한 방향벡터와 norm을 계산함으로써 특정 픽셀 영역에 포인트 추출이 몰리는 현상을 방지한다.

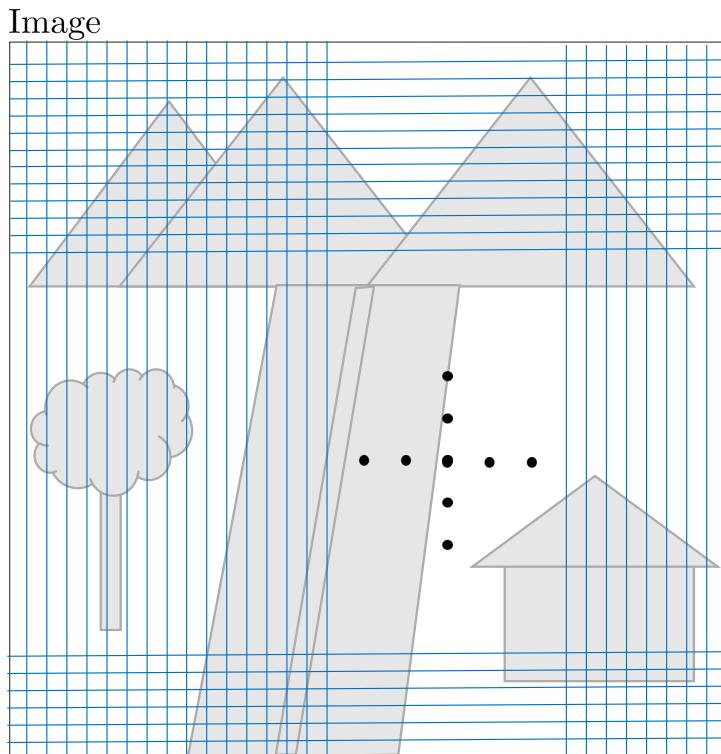
5.2. Pixel Selection → Dynamic Grid



Dynamic Grid

4) 만약 픽셀이 Smoothed threshold보다 작은 경우 $Lv0 \rightarrow Lv1 \rightarrow Lv2$ 순서로 값을 비교한다.
이 때 레벨이 높을수록 threshold에 0.75를 곱한 후 픽셀의 밝기 값과 비교하여 높은 피라미드
레벨의 픽셀일수록 추출될 확률을 높인다.

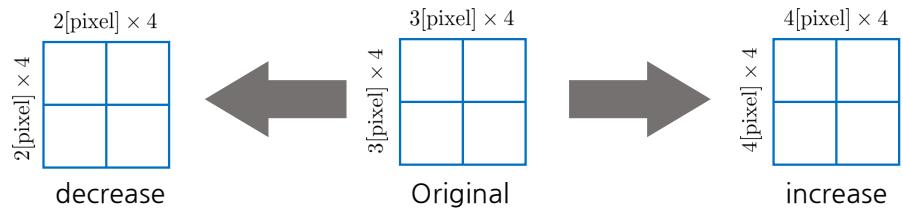
5.2. Pixel Selection → Dynamic Grid



Dynamic Grid

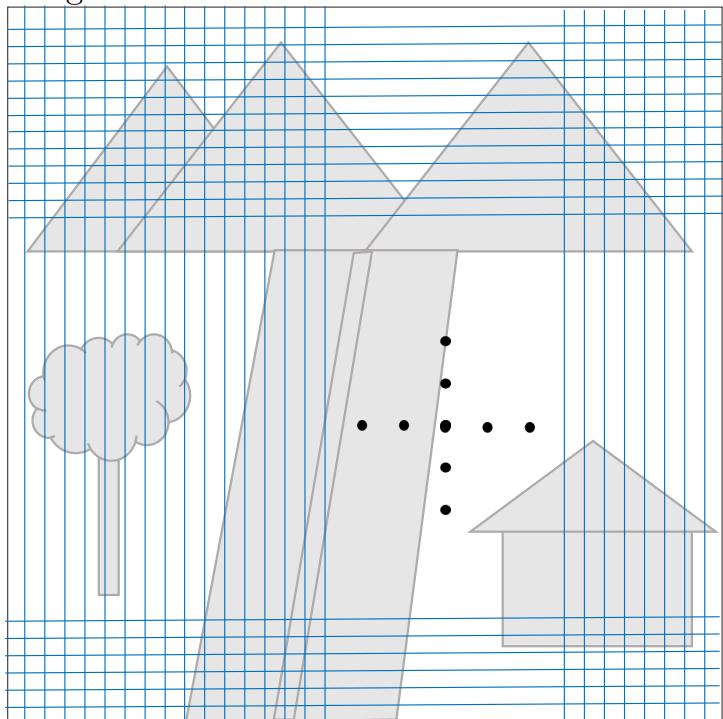
4) 만약 픽셀이 Smoothed threshold보다 작은 경우 $Lv0 \rightarrow Lv1 \rightarrow Lv2$ 순서로 값을 비교한다.
이 때 레벨이 높을수록 threshold에 0.75를 곱한 후 픽셀의 밝기 값과 비교하여 높은 피라미드
레벨의 픽셀일수록 추출될 확률을 높인다.

5) 모든 픽셀을 전부 검사했음에도 추출한 포인트의 총합이 원하는 총량보다 적은 경우
 $N_{want}/N_{have} > 1.25$ 에는 8×8 로 한 Grid의 크기를 줄인 후 다시 검색하고 만약 추출한 포인트가
너무 많은 경우 $N_{want}/N_{have} < 0.25$ 에는 16×16 의 Grid 크기로 포인트를 다시 추출한다.



5.2. Pixel Selection → Dynamic Grid

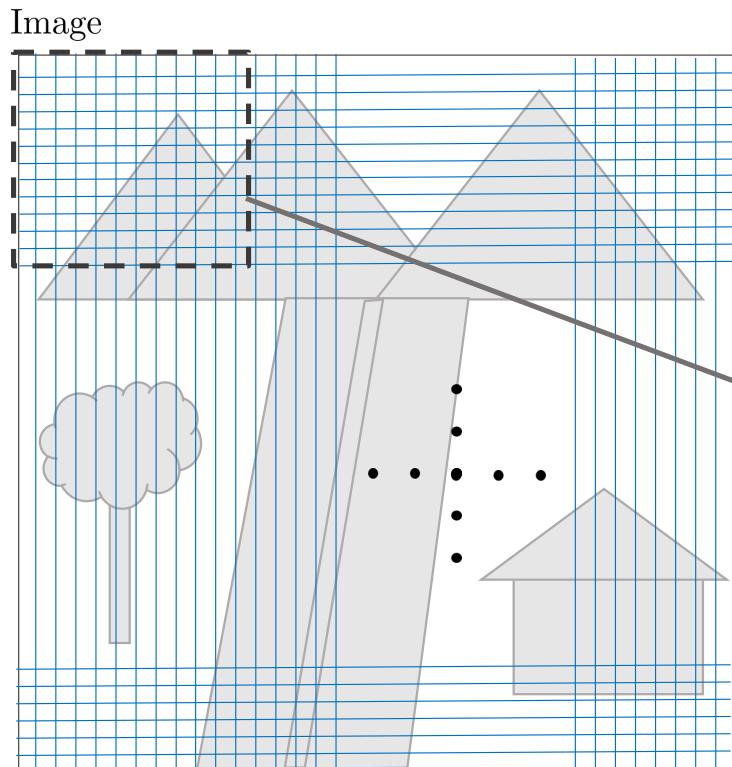
Image



Dynamic Grid

6) 한 이미지에서 모든 Grid에 대해 루프를 돌면서 Smoothed threshold를 통과한 픽셀들만 Pixel map에 저장한다.

5.2. Pixel Selection → Dynamic Grid



Dynamic Grid

6) 한 이미지에서 모든 Grid에 대해 루프를 돌면서 Smoothed threshold를 통과한 픽셀들만 Pixel map에 저장한다.

Pixel map에는 Lv0에서 추출된 포인트일 경우 1, Lv1에서 추출된 포인트는 2, Lv3에서 추출된 포인트는 4의 값이 해당 픽셀 위치에 저장된다. 포인트가 추출되지 않은 영역의 값은 저장되지 않는다. ($= 0$)

Pixel map

1	2	4			2	
	4		2			1
1		1	2			
			4			
	1	2				

모든 픽셀을 그리지 않고 12x12 Grid만 그린 그림
글씨가 Grid 영역보다 작은 이유는 글씨가 한 픽셀을 의미하기 때문

6. DSO Code Review

1.1. Initialization → Error Function Formulation

CoarseInitializer.cpp::calcResAndGS()::L385

```
For(int idx=0;idx<patternNum;idx++)
{
    int dx = patternP[idx][0];
    int dy = patternP[idx][1];

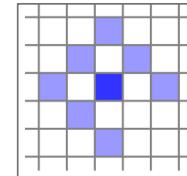
    Vec3f pt = RKi * Vec3f(point->u+dx, point->v+dy, 1) + t*point->idepth_new;
    float u = pt[0] / pt[2];
    float v = pt[1] / pt[2];
    float Ku = fxl * u + cxl;
    float Kv = fyl * v + cyl;
    float new_idepth = point->idepth_new / pt[2];

    if(!(Ku > 1 && Kv > 1 && Ku < wl-2 && Kv < hl-2 && new_idepth > 0))
    {
        isGood = false;
        break;
    }

    Vec3f hitColor = getInterpolatedElement33(colorNew, Ku, Kv, wl);
    //Vec3f hitColor = getInterpolatedElement33BiCub(colorNew, Ku, Kv, wl);
    //float rlr = colorRef[point->u+dx + (point->v+dy) * wl][0];
    float rlr = getInterpolatedElement31(colorRef, point->u+dx, point->v+dy, wl);

    if(!std::isfinite(rlr) || !std::isfinite((float)hitColor[0]))
    {
        isGood = false;
        break;
    }

    float residual = hitColor[0] - r2new_aff[0] * rlr - r2new_aff[1];
    float hw = fabs(residual) < setting_hubertH ? 1 : setting_hubertH / fabs(residual);
    energy += hw * residual * residual * (2 - hw);
}
```



패턴의 개수(8개)만큼 루프를 돌면서 각 점들의 residual을 계산한다.

1.1. Initialization → Error Function Formulation

CODE REVIEW

CoarseInitializer.cpp::calcResAndGS()::L385

```

for(int idx=0;idx<patternNum;idx++)
{
    int dx = patternP[idx][0];
    int dy = patternP[idx][1];

    Vec3f pt = RKi * Vec3f(point->u+dx, point->v+dy, 1) + t*point->idepth_new;
    float u = pt[0] / pt[2];
    float v = pt[1] / pt[2];
    float Ku = fxl * u + cxl;
    float Kv = fyl * v + cyl;
    float new_idepth = point->idepth_new/pt[2];

    if(!(Ku > 1 && Kv > 1 && Ku < wl-2 && Kv < hl-2 && new_idepth > 0))
    {
        isGood = false;
        break;
    }

    Vec3f hitColor = getInterpolatedElement33(colorNew, Ku, Kv, wl);
    //Vec3f hitColor = getInterpolatedElement33BiCub(colorNew, Ku, Kv, wl);
    //float rLR = colorRef[point->u+dx + (point->v+dy) * wl][0];
    float rLR = getInterpolatedElement31(colorRef, point->u+dx, point->v+dy, wl);

    if(!std::isfinite(rLR) || !std::isfinite((float)hitColor[0]))
    {
        isGood = false;
        break;
    }

    float residual = hitColor[0] - r2new_aff[0] * rLR - r2new_aff[1];
    float hw = fabs(residual) < setting_hubertH ? 1 : setting_hubertH / fabs(residual);
    energy += hw * residual*residual*(2-hw);
}

```

다음을 순서대로 계산한다.

$$\mathbf{X}_2 = \mathbf{R}\mathbf{K}^{-1}\left(\begin{bmatrix} u_1 \\ v_1 \\ 1 \end{bmatrix}\right) + \mathbf{t}\rho_1 = \begin{bmatrix} X_2 \\ Y_2 \\ Z_2 \end{bmatrix}$$

$$\bar{u}_2 = \frac{X_2}{Z_2}$$

$$\bar{v}_2 = \frac{Y_2}{Z_2}$$

$$u_2 = f_x \bar{x}_2 + c_x$$

$$v_2 = f_y \bar{v}_2 + c_y$$

$$\rho_2 = \frac{\rho_1}{Z_2} \leftarrow \text{해당 부분 불확실}$$

$$\mathbf{p}_1 = \pi(\mathbf{X}_1) = \mathbf{K}\rho_1\mathbf{X}_1$$

$$\mathbf{X}_1 = \pi^{-1}(\mathbf{p}_1) = \mathbf{K}^{-1}/\rho_1\mathbf{p}_1$$

$$\mathbf{X}_2 = \mathbf{T}\mathbf{X}_1 = \mathbf{R}(\mathbf{X}_1) + \mathbf{t}$$

1.1. Initialization → Error Function Formulation

CoarseInitializer.cpp::calcResAndGS()::L385

```

for(int idx=0;idx<patternNum;idx++)
{
    int dx = patternP[idx][0];
    int dy = patternP[idx][1];

    Vec3f pt = RKi * Vec3f(point->u+dx, point->v+dy, 1) + t*point->idepth_new;
    float u = pt[0] / pt[2];
    float v = pt[1] / pt[2];
    float Ku = fxl * u + cxl;           새로 계산한 픽셀이 이미지 내부에 있는지
    float Kv = fyl * v + cyl;
    float new_idepth = point->idepth_new/pt[2]; 새로 계산한 idepth 값이 0보다 큰지 확인

    if(!(Ku > 1 && Kv > 1 && Ku < wl-2 && Kv < hl-2 && new_idepth > 0))
    {
        isGood = false;
        break;
    }

    Vec3f hitColor = getInterpolatedElement33(colorNew, Ku, Kv, wl);
    //Vec3f hitColor = getInterpolatedElement33BiCub(colorNew, Ku, Kv, wl);
    //float rlr = colorRef[point->u+dx + (point->v+dy) * wl][0];
    float rlr = getInterpolatedElement31(colorRef, point->u+dx, point->v+dy, wl);

    if(!std::isfinite(rlr) || !std::isfinite((float)hitColor[0]))
    {
        isGood = false;
        break;
    }

    float residual = hitColor[0] - r2new_aff[0] * rlr - r2new_aff[1];
    float hw = fabs(residual) < setting_hubertH ? 1 : setting_hubertH / fabs(residual);
    energy += hw * residual*residual*(2-hw);
}

```

1.1. Initialization → Error Function Formulation

CODE REVIEW

CoarseInitializer.cpp::calcResAndGS()::L385

```
for(int idx=0;idx<patternNum;idx++)
{
    int dx = patternP[idx][0];
    int dy = patternP[idx][1];

    Vec3f pt = RKi * Vec3f(point->u+dx, point->v+dy, 1) + t*point->idepth_new;
    float u = pt[0] / pt[2];
    float v = pt[1] / pt[2];
    float Ku = fxl * u + cxl;
    float Kv = fyl * v + cyl;
    float new_idepth = point->idepth_new/pt[2];

    if(!(Ku > 1 && Kv > 1 && Ku < wl-2 && Kv < hl-2 && new_idepth > 0))
    {
        isGood = false;
        break;
    }

    Vec3f hitColor = getInterpolatedElement33(colorNew, Ku, Kv, wl);
    //Vec3f hitColor = getInterpolatedElement33BiCub(colorNew, Ku, Kv, wl);
    //float rlr = colorRef[point->u+dx + (point->v+dy) * wl][0];
    float rlr = getInterpolatedElement31(colorRef, point->u+dx, point->v+dy, wl);

    if(!std::isfinite(rlr) || !std::isfinite((float)hitColor[0]))
    {
        isGood = false;
        break;
    }

    float residual = hitColor[0] - r2new_aff[0] * rlr - r2new_aff[1];
    float hw = fabs(residual) < setting_hubertH ? 1 : setting_hubertH / fabs(residual);
    energy += hw * residual*residual*(2-hw);
```

새로운 이미지에서 Ku, Kv 픽셀 위치의 밝기 값 계산 $I_2(p_2)$

이전 이미지에서 현재 픽셀 위치의 밝기 값 계산 $I_1(p_1)$

hitColor[0]: $I_2(p_2)$

hitColor[1]: $\Delta I_{2,x}(p_2) = \Delta I_x$ for simplify

hitColor[2]: $\Delta I_{2,y}(p_2) = \Delta I_y$ for simplify

1.1. Initialization → Error Function Formulation

CoarseInitializer.cpp::calcResAndGS()::L385

```

for(int idx=0;idx<patternNum;idx++)
{
    int dx = patternP[idx][0];
    int dy = patternP[idx][1];

    Vec3f pt = RKi * Vec3f(point->u+dx, point->v+dy, 1) + t*point->idepth_new;
    float u = pt[0] / pt[2];
    float v = pt[1] / pt[2];
    float Ku = fxl * u + cxl;
    float Kv = fyl * v + cyl;
    float new_idepth = point->idepth_new/pt[2];

    if(!(Ku > 1 && Kv > 1 && Ku < wl-2 && Kv < hl-2 && new_idepth > 0))
    {
        isGood = false;
        break;
    }

    Vec3f hitColor = getInterpolatedElement33(colorNew, Ku, Kv, wl);
    //Vec3f hitColor = getInterpolatedElement33BiCub(colorNew, Ku, Kv, wl);
    //float rlr = colorRef[point->u+dx + (point->v+dy) * wl][0];
    float rlr = getInterpolatedElement31(colorRef, point->u+dx, point->v+dy, wl);

    if(!std::isfinite(rlr) || !std::isfinite((float)hitColor[0])) 밝기(intensity) 값이 유효한 값인지 확인
    {
        isGood = false;
        break;
    }

    float residual = hitColor[0] - r2new_aff[0] * rlr - r2new_aff[1];
    float hw = fabs(residual) < setting_hubertH ? 1 : setting_hubertH / fabs(residual);
    energy += hw * residual*residual*(2-hw);
}

```

1.1. Initialization → Error Function Formulation

CODE REVIEW

CoarseInitializer.cpp::calcResAndGS()::L385

```

for(int idx=0;idx<patternNum;idx++)
{
    int dx = patternP[idx][0];
    int dy = patternP[idx][1];

    Vec3f pt = RKi * Vec3f(point->u+dx, point->v+dy, 1) + t*point->idepth_new;
    float u = pt[0] / pt[2];
    float v = pt[1] / pt[2];
    float Ku = fxl * u + cxl;
    float Kv = fyl * v + cyl;
    float new_idepth = point->idepth_new/pt[2];

    if(!(Ku > 1 && Kv > 1 && Ku < wl-2 && Kv < hl-2 && new_idepth > 0))
    {
        isGood = false;
        break;
    }

    Vec3f hitColor = getInterpolatedElement33(colorNew, Ku, Kv, wl);
    //Vec3f hitColor = getInterpolatedElement33BiCub(colorNew, Ku, Kv, wl);
    //float rlR = colorRef[point->u+dx + (point->v+dy) * wl][0];
    float rlR = getInterpolatedElement31(colorRef, point->u+dx, point->v+dy, wl);

    if(!std::isfinite(rlR) || !std::isfinite((float)hitColor[0]))
    {
        isGood = false;
        break;
    }

    float residual = hitColor[0] - r2new_aff[0] * rlR - r2new_aff[1];
    float hw = fabs(residual) < setting_hubertH ? 1 : setting_hubertH / fabs(residual);
    energy += hw * residual*residual*(2-hw);
}

```

다음을 순서대로 계산한다.

$$\mathbf{r}(\mathbf{p}_1) = \mathbf{I}_2(\mathbf{p}_2) - \exp(a)\mathbf{I}_1(\mathbf{p}_1) - b$$

$$H(\mathbf{r}) = \begin{cases} \mathbf{r}^2/2, & |\mathbf{r}| < \sigma \\ \sigma(|\mathbf{r}| - \sigma/2) & |\mathbf{r}| \geq \sigma \end{cases}$$

$$w_h = \begin{cases} 1 & |\mathbf{r}| < \sigma \\ \sigma/|\mathbf{r}| & |\mathbf{r}| \geq \sigma \end{cases}$$

$$H(\mathbf{r}) = w_h \mathbf{r}^2 (1 - w_h/2)$$

(x2) 한 후 energy 변수에 저장되는 듯하다.

1.3. Initialization → Jacobian Derivation

CoarseInitializer.cpp::calcResAndGS():L424

```
float dxdx = (t[0]-t[2]*u)/pt[2];
float dydd = (t[1]-t[2]*v)/pt[2];

if(hw < 1) hw = sqrtf(hw);

float dxInterp = hw*hitColor[1]*fxl;
float dyInterp = hw*hitColor[2]*fyl;

dp0[idx] = new_idepth*dxInterp;
dp1[idx] = new_idepth*dyInterp;
dp2[idx] = -new_idepth*(u*dxInterp + v*dyInterp);
dp3[idx] = -u*v*dxInterp - (1+v)*dyInterp;
dp4[idx] = (1+u)*dxInterp + u*v*dyInterp;
dp5[idx] = -v*dxInterp + u*dyInterp;
dp6[idx] = - hw*r2new_aff[0] * rlR;
dp7[idx] = - hw*1;
dd[idx] = dxInterp * dxdx + dyInterp * dydd;
r[idx] = hw*residual;
```

다음을 순서대로 계산한다.

$$(t_x - t_z u'_2) \rho_2$$

$$(t_y - t_z v'_2) \rho_2$$

1.3. Initialization → Jacobian Derivation

CoarselInitializer.cpp::calcResAndGS()::L424

```

float dxd = (t[0]-t[2]*u)/pt[2];
float dydd = (t[1]-t[2]*v)/pt[2];

if(hw < 1) hw = sqrtf(hw); →  $w_h \rightarrow \sqrt{w_h}$ 

float dxInterp = hw*hitColor[1]*fxl;
float dyInterp = hw*hitColor[2]*fyl;

dp0[idx] = new_idepth*dxInterp;
dp1[idx] = new_idepth*dyInterp;
dp2[idx] = -new_idepth*(u*dxInterp + v*dyInterp);
dp3[idx] = -u*v*dxInterp - (1+v*v)*dyInterp;
dp4[idx] = (1+u*u)*dxInterp + u*v*dyInterp;
dp5[idx] = -v*dxInterp + u*dyInterp;
dp6[idx] = - hw*r2new_aff[0] * rlR;
dp7[idx] = - hw*1;
dd[idx] = dxInterp * dxd + dyInterp * dydd;
r[idx] = hw*residual;

```

1.3. Initialization → Jacobian Derivation

CoarseInitializer.cpp::calcResAndGS()::L424

```

float dxd = (t[0]-t[2]*u)/pt[2];
float dyd = (t[1]-t[2]*v)/pt[2];

if(hw < 1) hw = sqrtf(hw);

float dxInterp = hw*hitColor[1]*fxl;
float dyInterp = hw*hitColor[2]*fyl;

dp0[idx] = new_idepth*dxInterp;
dp1[idx] = new_idepth*dyInterp;
dp2[idx] = -new_idepth*(u*dxInterp + v*dyInterp);
dp3[idx] = -u*v*dxInterp - (1+v)*dyInterp;
dp4[idx] = (1+u)*dxInterp + u*v*dyInterp;
dp5[idx] = -v*dxInterp + u*dyInterp;
dp6[idx] = - hw*r2new_aff[0] * rlR;
dp7[idx] = - hw*1;
dd[idx] = dxInterp * dxd + dyInterp * dyd;
r[idx] = hw*residual;

```

다음을 순서대로 계산한다.

$$\sqrt{w_h} \Delta I_x f_x$$

$$\sqrt{w_h} \Delta I_y f_y$$

1.3. Initialization → Jacobian Derivation

CoarselInitializer.cpp::calcResAndGS()::L424

```

float dxd = (t[0]-t[2]*u)/pt[2];
float dyd = (t[1]-t[2]*v)/pt[2];

if(hw < 1) hw = sqrtf(hw);

float dxInterp = hw*hitColor[1]*fxl;
float dyInterp = hw*hitColor[2]*fyl;

dp0[idx] = new_idepth*dxInterp;
dp1[idx] = new_idepth*dyInterp;
dp2[idx] = -new_idepth*(u*dxInterp + v*dyInterp);
dp3[idx] = -u*v*dxInterp - (1+v)*v*dyInterp;
dp4[idx] = (1+u)*dxInterp + u*v*dyInterp;
dp5[idx] = -v*dxInterp + u*dyInterp;
apb[iax] = -hw*rznew_afr[t] * rIR;
dp7[idx] = - hw*1;
dd[idx] = dxInterp * dxd + dyInterp * dyd;
r[idx] = hw*residual;

```

$$\frac{\partial f(\mathbf{x})}{\partial \delta \xi} = \sqrt{w_h} \begin{bmatrix} \nabla \mathbf{I}_x \rho_2 f_x \\ \nabla \mathbf{I}_y \rho_2 f_y \\ -\rho_2 (\nabla \mathbf{I}_x f_x u'_2 + \nabla \mathbf{I}_y f_y v'_2) \\ -\nabla \mathbf{I}_x f_x u'_2 v'_2 - \nabla \mathbf{I}_y f_y (1 + v'^2_2) \\ \nabla \mathbf{I}_x f_x (1 + u'^2_2) + \nabla \mathbf{I}_y f_y u'_2 v'_2 \\ -\nabla \mathbf{I}_x f_x v'_2 + \nabla \mathbf{I}_y f_y u'_2 \end{bmatrix}^T$$

1.3. Initialization → Jacobian Derivation

CoarselInitializer.cpp::calcResAndGS()::L424

```

float dxdx = (t[0]-t[2]*u)/pt[2];
float dydd = (t[1]-t[2]*v)/pt[2];

if(hw < 1) hw = sqrtf(hw);

float dxInterp = hw*hitColor[1]*fxl;
float dyInterp = hw*hitColor[2]*fyl;

dp0[idx] = new_idepth*dxInterp;
dp1[idx] = new_idepth*dyInterp;
dp2[idx] = -new_idepth*(u*dxInterp + v*dyInterp);
dp3[idx] = -u*v*dxInterp - (1+v*v)*dyInterp;
dp4[idx] = (1+u*u)*dxInterp + u*v*dyInterp;
dp5[idx] = -v*dxInterp + u*dyInterp;
dp6[idx] = - hw*r2new_aff[0] * rlR;
dp7[idx] = - hw*1;
du[idx] = dxInterp * dxdx + dyInterp * dydd;
r[idx] = hw*residual;

```

$$\frac{\partial f(\mathbf{x})}{\partial a} = -\sqrt{w_h} \exp(a) I_1(\mathbf{p}_1)$$

$$\frac{\partial f(\mathbf{x})}{\partial b} = -\sqrt{w_h}$$

1.3. Initialization → Jacobian Derivation

CoarseInitializer.cpp::calcResAndGS()::L424

```
float dxd = (t[0]-t[2]*u)/pt[2];
float dydd = (t[1]-t[2]*v)/pt[2];

if(hw < 1) hw = sqrtf(hw);

float dxInterp = hw*hitColor[1]*fxl;
float dyInterp = hw*hitColor[2]*fyl;

dp0[idx] = new_idepth*dxInterp;
dp1[idx] = new_idepth*dyInterp;
dp2[idx] = -new_idepth*(u*dxInterp + v*dyInterp);
dp3[idx] = -u*v*dxInterp - (1+v)*v*dyInterp;
dp4[idx] = (1+u)*dxInterp + u*v*dyInterp;
dp5[idx] = -v*dxInterp + u*dyInterp;
dp6[idx] = - hw*r2new_aff[0] * rlR;
dp7[idx] = - hw*1;

dd[idx] = dxInterp * dxd + dyInterp * dydd;
r_lax = nw_residual;
```

$$(t_x - t_z u'_2) \rho_2$$

$$(t_y - t_z v'_2) \rho_2$$

$$\sqrt{w_h} \Delta \mathbf{I}_x f_x$$

$$\sqrt{w_h} \Delta \mathbf{I}_y f_y$$

$$\frac{\partial f(\mathbf{x})}{\partial \rho_1} = \sqrt{w_h} \rho_1^{-1} \rho_2 (\nabla \mathbf{I}_x f_x (t_x - u'_2 t_z) + \nabla \mathbf{I}_y f_y (t_y - v'_2 t_z))$$

I can't find where ρ_1^{-1} is...

1.3. Initialization → Jacobian Derivation

CODE REVIEW

CoarselInitializer.cpp::calcResAndGS()::L424

```
float dxdx = (t[0]-t[2]*u)/pt[2];
float dydd = (t[1]-t[2]*v)/pt[2];

if(hw < 1) hw = sqrtf(hw);

float dxInterp = hw*hitColor[1]*fxl;
float dyInterp = hw*hitColor[2]*fyl;

dp0[idx] = new_idepth*dxInterp;
dp1[idx] = new_idepth*dyInterp;
dp2[idx] = -new_idepth*(u*dxInterp + v*dyInterp);
dp3[idx] = -u*v*dxInterp - (1+v)*dyInterp;
dp4[idx] = (1+u)*dxInterp + u*v*dyInterp;
dp5[idx] = -v*dxInterp + u*dyInterp;
dp6[idx] = - hw*r2new_aff[0] * rlR;
dp7[idx] = - hw*1;
dd[idx] = dxInterp * dxdx + dyInterp * dydd;
r[idx] = hw*residual;
```

$$f(\mathbf{x}) = \sqrt{w_h} \mathbf{r}$$

RECAP

$$f(\mathbf{x})^T f(\mathbf{x}) = H(\mathbf{r})$$

$$f(\mathbf{x}) = \sqrt{w_h} \mathbf{r} = \sqrt{w_h} (\mathbf{I}_2(\mathbf{p}_2) - \exp(a)\mathbf{I}_1(\mathbf{p}_1) - b)$$

2.1. Frames → Pose Tracking

CoarseTracker.cpp::calcRes()::L450

```

/// calculate the residual.
float refColor = lpc_color[i];
Vec3f hitColor = getInterpolatedElement33(dINewl, Ku, Kv, wl); // interpolate on new frame.
if(!std::isfinite((float)hitColor[0])) continue;
float residual = hitColor[0] - (float)(affLL[0] * refColor + affLL[1]);
// Huber weight
float hw = fabs(residual) < setting_huberTH ? 1 : setting_huberTH / fabs(residual);

if(fabs(residual) > cutoffTH)
{
    if(debugPlot) resImage->setPixel4(lpc_u[i], lpc_v[i], Vec3b(0,0,255));
    E += maxEnergy; // energy value.
    numTermsInE++; // number in E.
    numSaturated++; // greater than threshold number.
}
else {
    if(debugPlot) {
        resImage->setPixel4(lpc_u[i], lpc_v[i], Vec3b(residual+128,residual+128,residual+128));
    }

    E += hw * residual*residual*(2-hw);
    numTermsInE++;
    buf_warped_idepth[numTermsInWarped] = new_idepth;
    buf_warped_u[numTermsInWarped] = u;
    buf_warped_v[numTermsInWarped] = v;
    buf_warped_dx[numTermsInWarped] = hitColor[1];
    buf_warped_dy[numTermsInWarped] = hitColor[2];
    buf_warped_residual[numTermsInWarped] = residual;
    buf_warped_weight[numTermsInWarped] = hw;
    buf_warped_refColor[numTermsInWarped] = lpc_color[i];
    numTermsInWarped++;
}

```

2.1. Frames → Pose Tracking

CoarseTracker.cpp::calcRes()::L450

```

/// calculate the residual.
float refColor = lpc_color[i];
Vec3f hitColor = getInterpolatedElement33(dINewl, Ku, Kv, wl); // interpolate on new frame
if(!std::isfinite((float)hitColor[0])) continue;
float residual = hitColor[0] - (float)(affLL[0] * refColor + affLL[1]);
// Huber weight
float hw = fabs(residual) < setting_huberTH ? 1 : setting_huberTH / fabs(residual);

if(fabs(residual) > cutoffTH)
{
    if(debugPlot) resImage->setPixel4(lpc_u[i], lpc_v[i], Vec3b(0,0,255));
    E += maxEnergy; // energy value.
    numTermsInE++; // number in E.
    numSaturated++; // greater than threshold number.
}
else {
    if(debugPlot) {
        resImage->setPixel4(lpc_u[i], lpc_v[i], Vec3b(residual+128,residual+128,residual+128));
    }

    E += hw * residual*residual*(2-hw);
    numTermsInE++;
    buf_warped_idepth[numTermsInWarped] = new_idepth;
    buf_warped_u[numTermsInWarped] = u;
    buf_warped_v[numTermsInWarped] = v;
    buf_warped_dx[numTermsInWarped] = hitColor[1];
    buf_warped_dy[numTermsInWarped] = hitColor[2];
    buf_warped_residual[numTermsInWarped] = residual;
    buf_warped_weight[numTermsInWarped] = hw;
    buf_warped_refColor[numTermsInWarped] = lpc_color[i];
    numTermsInWarped++;
}

```

이전 이미지에서 현재 픽셀 위치의 밝기 값 계산 $I_1(p_1)$

새로운 이미지에서 Ku, Kv 픽셀 위치의 밝기 값 계산 $I_2(p_2)$

hitColor[0]: $I_2(p_2)$

hitColor[1]: $\Delta I_{2,x}(p_2) = \Delta I_x$ for simplify

hitColor[2]: $\Delta I_{2,y}(p_2) = \Delta I_y$ for simplify

2.1. Frames → Pose Tracking

CoarseTracker.cpp::calcRes()::L450

```

/// calculate the residual.
float refColor = lpc_color[i];           hitColor 값이 유효한 값이 아닐 경우 다음 점으로 이동
Vec3f hitColor = getInterpolatedElement33(dINewl, Ku, Kv, wl); // interpolate on new frame.
if(!std::isfinite((float)hitColor[0])) continue;
float residual = hitColor[0] - (float)(affLL[0] * refColor + affLL[1]);
// Huber weight
float hw = fabs(residual) < setting_huberTH ? 1 : setting_huberTH / fabs(residual);

if(fabs(residual) > cutoffTH)
{
    if(debugPlot) resImage->setPixel4(lpc_u[i], lpc_v[i], Vec3b(0,0,255));
    E += maxEnergy; // energy value.
    numTermsInE++; // number in E.
    numSaturated++; // greater than threshold number.
}
else {
    if(debugPlot) {
        resImage->setPixel4(lpc_u[i], lpc_v[i], Vec3b(residual+128,residual+128,residual+128));
    }

    E += hw * residual*residual*(2-hw);
    numTermsInE++;
    buf_warped_idepth[numTermsInWarped] = new_idepth;
    buf_warped_u[numTermsInWarped] = u;
    buf_warped_v[numTermsInWarped] = v;
    buf_warped_dx[numTermsInWarped] = hitColor[1];
    buf_warped_dy[numTermsInWarped] = hitColor[2];
    buf_warped_residual[numTermsInWarped] = residual;
    buf_warped_weight[numTermsInWarped] = hw;
    buf_warped_refColor[numTermsInWarped] = lpc_color[i];
    numTermsInWarped++;
}

```

2.1. Frames → Pose Tracking

CODE REVIEW

CoarseTracker.cpp::calcRes()::L450

```

/// calculate the residual.
float refColor = lpc_color[i];
Vec3f hitColor = getInterpolatedElement33(dINewl, Ku, Kv, wl); // interpolate on new frame.
if(!std::isfinite((float)hitColor[0])) continue;
float residual = hitColor[0] - (float)(affLL[0] * refColor + affLL[1]);
// Huber weight
float hw = fabs(residual) < setting_huberTH ? 1 : setting_huberTH / fabs(residual);

if(fabs(residual) > cutoffTH)
{
    if(debugPlot) resImage->setPixel4(lpc_u[i], lpc_v[i], Vec3b(0,0,255));
    E += maxEnergy; // energy value.
    numTermsInE++; // number in E.
    numSaturated++; // greater than threshold number.
}
else {
    if(debugPlot) {
        resImage->setPixel4(lpc_u[i], lpc_v[i], Vec3b(residual+128,residual+128,residual+128));
    }

    E += hw * residual*residual*(2-hw);
    numTermsInE++;
    buf_warped_idepth[numTermsInWarped] = new_idepth;
    buf_warped_u[numTermsInWarped] = u;
    buf_warped_v[numTermsInWarped] = v;
    buf_warped_dx[numTermsInWarped] = hitColor[1];
    buf_warped_dy[numTermsInWarped] = hitColor[2];
    buf_warped_residual[numTermsInWarped] = residual;
    buf_warped_weight[numTermsInWarped] = hw;
    buf_warped_refColor[numTermsInWarped] = lpc_color[i];
    numTermsInWarped++;
}

```

다음을 순서대로 계산한다.

$$\mathbf{r}(\mathbf{p}_1) = \mathbf{I}_2(\mathbf{p}_2) - \exp(a)\mathbf{I}_1(\mathbf{p}_1) - b$$

$$H(\mathbf{r}) = \begin{cases} \mathbf{r}^2/2, & |\mathbf{r}| < \sigma \\ \sigma(|\mathbf{r}| - \sigma/2) & |\mathbf{r}| \geq \sigma \end{cases}$$

$$w_h = \begin{cases} 1 & |\mathbf{r}| < \sigma \\ \sigma/|\mathbf{r}| & |\mathbf{r}| \geq \sigma \end{cases}$$

2.1. Frames → Pose Tracking

CODE REVIEW

CoarseTracker.cpp::calcRes()::L450

```
/// calculate the residual.
float refColor = lpc_color[i];
Vec3f hitColor = getInterpolatedElement33(dINewl, Ku, Kv, wl); // interpolate on new frame.
if(!std::isfinite((float)hitColor[0])) continue;
float residual = hitColor[0] - (float)(affLL[0] * refColor + affLL[1]);
// Huber weight
float hw = fabs(residual) < setting_hubertH ? 1 : setting_hubertH / fabs(residual);

if(fabs(residual) > cutoffTH)
{
    if(debugPlot) resImage->setPixel4(lpc_u[i], lpc_v[i], Vec3b(0,0,255));
    E += maxEnergy; // energy value.
    numTermsInE++; // number in E.
    numSaturated++; // greater than threshold number.
}
else {
    if(debugPlot) {
        resImage->setPixel4(lpc_u[i], lpc_v[i], Vec3b(residual+128,residual+128,residual+128));
    }

    E += hw * residual*residual*(2-hw);
    numTermsInE++;
    buf_warped_idepth[numTermsInWarped] = new_idepth;
    buf_warped_u[numTermsInWarped] = u;
    buf_warped_v[numTermsInWarped] = v;
    buf_warped_dx[numTermsInWarped] = hitColor[1];
    buf_warped_dy[numTermsInWarped] = hitColor[2];
    buf_warped_residual[numTermsInWarped] = residual;
    buf_warped_weight[numTermsInWarped] = hw;
    buf_warped_refColor[numTermsInWarped] = lpc_color[i];
    numTermsInWarped++;
}
```

residual 값이 특정 기준보다 큰 경우 Huber functio에 linear 영향을 받기 위해
에너지를 증가시키고 해당 경우는 Skip

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/CoarseTracker.cpp#L450>

2.1. Frames → Pose Tracking

CoarseTracker.cpp::calcRes()::L450

```

/// calculate the residual.
float refColor = lpc_color[i];
Vec3f hitColor = getInterpolatedElement33(dINewl, Ku, Kv, wl); // interpolate on new frame.
if(!std::isfinite((float)hitColor[0])) continue;
float residual = hitColor[0] - (float)(affLL[0] * refColor + affLL[1]);
// Huber weight
float hw = fabs(residual) < setting_huberTH ? 1 : setting_huberTH / fabs(residual);

if(fabs(residual) > cutoffTH)
{
    if(debugPlot) resImage->setPixel4(lpc_u[i], lpc_v[i], Vec3b(0,0,255));
    E += maxEnergy; // energy value.
    numTermsInE++; // number in E.
    numSaturated++; // greater than threshold number.
}
else {
    if(debugPlot) {
        resImage->setPixel4(lpc_u[i], lpc_v[i], Vec3b(residual+128,residual+128,residual+128));
    }

    E += hw * residual*residual*(2-hw);
    numTermsInE++;
    buf_warped_idepth[numTermsInWarped] = new_idepth;
    buf_warped_u[numTermsInWarped] = u;
    buf_warped_v[numTermsInWarped] = v;
    buf_warped_dx[numTermsInWarped] = hitColor[1];
    buf_warped_dy[numTermsInWarped] = hitColor[2];
    buf_warped_residual[numTermsInWarped] = residual;
    buf_warped_weight[numTermsInWarped] = hw;
    buf_warped_refColor[numTermsInWarped] = lpc_color[i];
    numTermsInWarped++;
}

```

residual 값이 특정 기준보다 작은 경우

2.1. Frames → Pose Tracking

CODE REVIEW

CoarseTracker.cpp::calcRes()::L450

```

/// calculate the residual.
float refColor = lpc_color[i];
Vec3f hitColor = getInterpolatedElement33(dINewl, Ku, Kv, wl); // interpolate on new frame.
if(!std::isfinite((float)hitColor[0])) continue;
float residual = hitColor[0] - (float)(affLL[0] * refColor + affLL[1]);
// Huber weight
float hw = fabs(residual) < setting_huberTH ? 1 : setting_huberTH / fabs(residual);

if(fabs(residual) > cutoffTH)
{
    if(debugPlot) resImage->setPixel4(lpc_u[i], lpc_v[i], Vec3b(0,0,255));
    E += maxEnergy; // energy value.
    numTermsInE++; // number in E.
    numSaturated++; // greater than threshold number.
}
else {
    if(debugPlot) {
        resImage->setPixel4(lpc_u[i], lpc_v[i], Vec3b(residual+128,residual+128,residual+128));
    }

    E += hw * residual*residual*(2-hw);
    numTermsInE++;
    buf_warped_idepth[numTermsInWarped] = new_idepth;
    buf_warped_u[numTermsInWarped] = u;
    buf_warped_v[numTermsInWarped] = v;
    buf_warped_dx[numTermsInWarped] = hitColor[1];
    buf_warped_dy[numTermsInWarped] = hitColor[2];
    buf_warped_residual[numTermsInWarped] = residual;
    buf_warped_weight[numTermsInWarped] = hw;
    buf_warped_refColor[numTermsInWarped] = lpc_color[i];
    numTermsInWarped++;
}

```

$$H(r) = w_h r^2 (1 - w_h/2)$$

(x2) 한 후 energy 변수에 저장되는 듯하다.

2.1. Frames → Pose Tracking

CoarseTracker.cpp::calcRes()::L450

```

/// calculate the residual.
float refColor = lpc_color[i];
Vec3f hitColor = getInterpolatedElement33(dINewl, Ku, Kv, wl); // interpolate on new frame.
if(!std::isfinite((float)hitColor[0])) continue;
float residual = hitColor[0] - (float)(affLL[0] * refColor + affLL[1]);
// Huber weight
float hw = fabs(residual) < setting_huberTH ? 1 : setting_huberTH / fabs(residual);

if(fabs(residual) > cutoffTH)
{
    if(debugPlot) resImage->setPixel4(lpc_u[i], lpc_v[i], Vec3b(0,0,255));
    E += maxEnergy; // energy value.
    numTermsInE++; // number in E.
    numSaturated++; // greater than threshold number.
}
else {
    if(debugPlot) {
        resImage->setPixel4(lpc_u[i], lpc_v[i], Vec3b(residual+128,residual+128,residual+128));
    }

    E += hw * residual*residual*(2-hw);      numTermsInE: number of projected point.
    numTermsInE++;
    buf_warped_idepth[numTermsInWarped] = new_idepth;
    buf_warped_u[numTermsInWarped] = u;
    buf_warped_v[numTermsInWarped] = v;
    buf_warped_dx[numTermsInWarped] = hitColor[1];
    buf_warped_dy[numTermsInWarped] = hitColor[2];
    buf_warped_residual[numTermsInWarped] = residual;
    buf_warped_weight[numTermsInWarped] = hw;
    buf_warped_refColor[numTermsInWarped] = lpc_color[i];
    numTermsInWarped++;
}

```

2.1. Frames → Pose Tracking

CoarseTracker.cpp::calcRes()::L450

```

/// calculate the residual.
float refColor = lpc_color[i];
Vec3f hitColor = getInterpolatedElement33(dINewl, Ku, Kv, wl); // interpolate on new frame.
if(!std::isfinite((float)hitColor[0])) continue;
float residual = hitColor[0] - (float)(affLL[0] * refColor + affLL[1]);
// Huber weight
float hw = fabs(residual) < setting_huberTH ? 1 : setting_huberTH / fabs(residual);

if(fabs(residual) > cutoffTH)
{
    if(debugPlot) resImage->setPixel4(lpc_u[i], lpc_v[i], Vec3b(0,0,255));
    E += maxEnergy; // energy value.
    numTermsInE++; // number in E.
    numSaturated++; // greater than threshold number.
}
else {
    if(debugPlot) {
        resImage->setPixel4(lpc_u[i], lpc_v[i], Vec3b(residual+128,residual+128,residual+128));
    }

    E += hw * residual*residual*(2-hw);
    numTermsInE++;
    buf_warped_idepth[numTermsInWarped] = new_idepth;
    buf_warped_u[numTermsInWarped] = u;
    buf_warped_v[numTermsInWarped] = v;
    buf_warped_dx[numTermsInWarped] = hitColor[1];
    buf_warped_dy[numTermsInWarped] = hitColor[2];
    buf_warped_residual[numTermsInWarped] = residual;
    buf_warped_weight[numTermsInWarped] = hw;
    buf_warped_refColor[numTermsInWarped] = lpc_color[i];
    numTermsInWarped++;
}

```

ρ_2
 u_2
 v_2
 ΔI_x
 ΔI_y
 r
 $H(r)$
 $I_1(p_1)$

순서대로 버퍼에 저장한다

2.1. Frames → Pose Tracking

CoarseTracker.cpp::calcGSSSE():L320

```

acc.updateSSE_eighted(
    _mm_mul_ps(id,dx),
    _mm_mul_ps(id,dy),
    _mm_sub_ps(zero, _mm_mul_ps(id,_mm_add_ps(_mm_mul_ps(u,dx),
    _mm_mul_ps(v,dy)))),
    _mm_sub_ps(zero, _mm_add_ps(
    _mm_mul_ps(_mm_mul_ps(u,v),dx),
    _mm_mul_ps(dy,_mm_add_ps(one, _mm_mul_ps(v,v))))),
    _mm_add_ps(
    _mm_mul_ps(_mm_mul_ps(u,v),dy),
    _mm_mul_ps(dx,_mm_add_ps(one, _mm_mul_ps(u,u)))),
    _mm_sub_ps(_mm_mul_ps(u,dy), _mm_mul_ps(v,dx)),
    _mm_mul_ps(a,_mm_sub_ps(b0, _mm_load_ps(buf_warped_refColor+i))),
    minusOne,
    _mm_load_ps(buf_warped_residual+i),
    _mm_load_ps(buf_warped_weight+i));

acc.finish();
H_out = acc.H.topLeftCorner<8,8>().cast<double>() * (1.0f/n);
b_out = acc.H.topRightCorner<8,1>().cast<double>() * (1.0f/n);

H_out.block<8,3>(0,0) *= SCALE_XI_ROT;
H_out.block<8,3>(0,3) *= SCALE_XI_TRANS;
H_out.block<8,1>(0,6) *= SCALE_A;
H_out.block<8,1>(0,7) *= SCALE_B;
H_out.block<3,8>(0,0) *= SCALE_XI_ROT;
H_out.block<3,8>(3,0) *= SCALE_XI_TRANS;
H_out.block<1,8>(6,0) *= SCALE_A;
H_out.block<1,8>(7,0) *= SCALE_B;
b_out.segment<3>(0) *= SCALE_XI_ROT;
b_out.segment<3>(3) *= SCALE_XI_TRANS;
b_out.segment<1>(6) *= SCALE_A;
b_out.segment<1>(7) *= SCALE_B;

```

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/CoarseTracker.cpp#L320>

2.1. Frames → Pose Tracking

CoarseTracker.cpp::calcGSSSE():L320

```

acc.updateSSE_eighted(
    _mm_mul_ps(id,dx),
    _mm_mul_ps(id,dy),
    _mm_sub_ps(zero, _mm_mul_ps(id,_mm_add_ps(_mm_mul_ps(u,dx),
    _mm_mul_ps(v,dy)))),
    _mm_sub_ps(zero, _mm_add_ps(
    _mm_mul_ps(_mm_mul_ps(u,v),dx),
    _mm_mul_ps(dy,_mm_add_ps(one, _mm_mul_ps(v,v))))),
    _mm_add_ps(
    _mm_mul_ps(_mm_mul_ps(u,v),dy),
    _mm_mul_ps(dx,_mm_add_ps(one, _mm_mul_ps(u,u)))),
    _mm_sub_ps(_mm_mul_ps(u,dy), _mm_mul_ps(v,dx)),
    _mm_mul_ps(a,_mm_sub_ps(b0, _mm_load_ps(buf_warped_refColor+i))),
    minusOne,
    _mm_load_ps(buf_warped_residual+i),
    _mm_load_ps(buf_warped_weight+i));
}

acc.finish();
H_out = acc.H.topLeftCorner<8,8>().cast<double>() * (1.0f/n);
b_out = acc.H.topRightCorner<8,1>().cast<double>() * (1.0f/n);

H_out.block<8,3>(0,0) *= SCALE_XI_ROT;
H_out.block<8,3>(0,3) *= SCALE_XI_TRANS;
H_out.block<8,1>(0,6) *= SCALE_A;
H_out.block<8,1>(0,7) *= SCALE_B;
H_out.block<3,8>(0,0) *= SCALE_XI_ROT;
H_out.block<3,8>(3,0) *= SCALE_XI_TRANS;
H_out.block<1,8>(6,0) *= SCALE_A;
H_out.block<1,8>(7,0) *= SCALE_B;
b_out.segment<3>(0) *= SCALE_XI_ROT;
b_out.segment<3>(3) *= SCALE_XI_TRANS;
b_out.segment<1>(6) *= SCALE_A;
b_out.segment<1>(7) *= SCALE_B;

```

$$\frac{\partial f(\mathbf{x})}{\partial \delta\xi} = \sqrt{w_h} \begin{bmatrix} \nabla_{\mathbf{I}_x} \rho_2 f_x \\ \nabla_{\mathbf{I}_y} \rho_2 f_y \\ -\rho_2 (\nabla_{\mathbf{I}_x} f_x u_2 + \nabla_{\mathbf{I}_y} f_y v_2) \\ -\nabla_{\mathbf{I}_x} f_x u_2 v_2 - \nabla_{\mathbf{I}_y} f_y (1 + v_2'^2) \\ \nabla_{\mathbf{I}_x} f_x (1 + u_2'^2) + \nabla_{\mathbf{I}_y} f_y u_2' v_2' \\ -\nabla_{\mathbf{I}_x} f_x v_2 + \nabla_{\mathbf{I}_y} f_y u_2 \end{bmatrix}^T$$

$$\frac{\partial f(\mathbf{x})}{\partial a} = \sqrt{w_h} \exp(a) \mathbf{I}_1(b_0 - \mathbf{p}_1)$$

$$\frac{\partial f(\mathbf{x})}{\partial b} = -\sqrt{w_h}$$

to construct $\mathbf{J}_\xi = [\frac{\partial f(\mathbf{x})}{\partial \delta\xi} \quad \frac{\partial f(\mathbf{x})}{\partial a} \quad \frac{\partial f(\mathbf{x})}{\partial b}]_{8 \times 1}$

2.1. Frames → Pose Tracking

CoarseTracker.cpp::calcGSSSE():L320

```

acc.updateSSE_eighted(
    _mm_mul_ps(id,dx),
    _mm_mul_ps(id,dy),
    _mm_sub_ps(zero, _mm_mul_ps(id,_mm_add_ps(_mm_mul_ps(u,dx),
    _mm_mul_ps(v,dy)))),
    _mm_sub_ps(zero, _mm_add_ps(
    _mm_mul_ps(_mm_mul_ps(u,v),dx),
    _mm_mul_ps(dy,_mm_add_ps(one, _mm_mul_ps(v,v))))),
    _mm_add_ps(
    _mm_mul_ps(_mm_mul_ps(u,v),dy),
    _mm_mul_ps(dx,_mm_add_ps(one, _mm_mul_ps(u,u)))),
    _mm_sub_ps(_mm_mul_ps(u,dy), _mm_mul_ps(v,dx)),
    _mm_mul_ps(a,_mm_sub_ps(b0, _mm_load_ps(buf_warped_refColor+i))),
    minusOne,
    _mm_load_ps(buf_warped_residual+i),
    _mm_load_ps(buf_warped_weight+i));
```



```

acc.finish();
H_out = acc.H.topLeftCorner<8,8>().cast<double>() * (1.0f/n);
b_out = acc.H.topRightCorner<8,1>().cast<double>() * (1.0f/n);

H_out.block<8,3>(0,0) *= SCALE_XI_ROT;
H_out.block<8,3>(0,3) *= SCALE_XI_TRANS;
H_out.block<8,1>(0,6) *= SCALE_A;
H_out.block<8,1>(0,7) *= SCALE_B;
H_out.block<3,8>(0,0) *= SCALE_XI_ROT;
H_out.block<3,8>(3,0) *= SCALE_XI_TRANS;
H_out.block<1,8>(6,0) *= SCALE_A;
H_out.block<1,8>(7,0) *= SCALE_B;
b_out.segment<3>(0) *= SCALE_XI_ROT;
b_out.segment<3>(3) *= SCALE_XI_TRANS;
b_out.segment<1>(6) *= SCALE_A;
b_out.segment<1>(7) *= SCALE_B;
```

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/CoarseTracker.cpp#L320>

2.1. Frames → Pose Tracking

CoarseTracker.cpp::calcGSSSE():L320

```

acc.updateSSE_eighted(
    _mm_mul_ps(id,dx),
    _mm_mul_ps(id,dy),
    _mm_sub_ps(zero, _mm_mul_ps(id,_mm_add_ps(_mm_mul_ps(u,dx),
    _mm_mul_ps(v,dy)))),
    _mm_sub_ps(zero, _mm_add_ps(
    _mm_mul_ps(_mm_mul_ps(u,v),dx),
    _mm_mul_ps(dy,_mm_add_ps(one, _mm_mul_ps(v,v))))),
    _mm_add_ps(
    _mm_mul_ps(_mm_mul_ps(u,v),dy),
    _mm_mul_ps(dx,_mm_add_ps(one, _mm_mul_ps(u,u)))),
    _mm_sub_ps(_mm_mul_ps(u,dy), _mm_mul_ps(v,dx)),
    _mm_mul_ps(a,_mm_sub_ps(b0, _mm_load_ps(buf_warped_refColor+i))),
    minusOne,
    _mm_load_ps(buf_warped_residual+i),  r
    _mm_load_ps(buf_warped_weight+i));  H(r)

acc.finish();
H_out = acc.H.topLeftCorner<8,8>().cast<double>() * (1.0f/n);
b_out = acc.H.topRightCorner<8,1>().cast<double>() * (1.0f/n);

H_out.block<8,3>(0,0) *= SCALE_XI_ROT;
H_out.block<8,3>(0,3) *= SCALE_XI_TRANS;
H_out.block<8,1>(0,6) *= SCALE_A;
H_out.block<8,1>(0,7) *= SCALE_B;
H_out.block<3,8>(0,0) *= SCALE_XI_ROT;
H_out.block<3,8>(3,0) *= SCALE_XI_TRANS;
H_out.block<1,8>(6,0) *= SCALE_A;
H_out.block<1,8>(7,0) *= SCALE_B;
b_out.segment<3>(0) *= SCALE_XI_ROT;
b_out.segment<3>(3) *= SCALE_XI_TRANS;
b_out.segment<1>(6) *= SCALE_A;
b_out.segment<1>(7) *= SCALE_B;

```

$$\mathbf{H}_{\xi\xi} = \sum \mathbf{J}_\xi^T \mathbf{J}_\xi$$

$$\mathbf{b}_\xi = \sum \mathbf{J}_\xi^T f(\mathbf{x})$$

2.1. Frames → Pose Tracking

CODE REVIEW

CoarseTracker.cpp::TrackNewestCoarse()::L536

```
int maxIterations[] = {10, 20, 50, 50, 50};
for(int lvl=coarsestLvl; lvl>=0; lvl--)
{
    Mat88 H; Vec8 b;
    float levelCutoffRepeat=1;
    Vec6 resOld = calcRes(lvl, refToNew_current, aff_g2l_current, setting_coarseCutoffTH*levelCutoffRepeat);
    while(resOld[5] > 0.6 && levelCutoffRepeat < 50)
    {
        levelCutoffRepeat*=2;
        resOld = calcRes(lvl, refToNew_current, aff_g2l_current, setting_coarseCutoffTH*levelCutoffRepeat);
        if(!setting_debugout_runquiet)
            printf("INCREASING cutoff to %f (ratio is %f)!\n", setting_coarseCutoffTH*levelCutoffRepeat, resOld[5]);
    }
    calcGSSSE(lvl, H, b, refToNew_current, aff_g2l_current);
    float lambda = 0.01;
    for(int iteration=0; iteration < maxIterations[lvl]; iteration++)
    {
        Mat88 Hl = H;
        for(int i=0;i<8;i++) Hl(i,i) *= (1+lambda);
        Vec8 inc = Hl.ldlt().solve(-b);
        {...}
        float extrapFac = 1;
        if(lambda < lambdaExtrapolationLimit) extrapFac = sqrt(sqrt(lambdaExtrapolationLimit / lambda));
        inc *= extrapFac;
        Vec8 incScaled = inc;
        incScaled.segment<3>(0) *= SCALE_XI_ROT;
        incScaled.segment<3>(3) *= SCALE_XI_TRANS;
        incScaled.segment<1>(6) *= SCALE_A;
        incScaled.segment<1>(7) *= SCALE_B;
        if(!std::isfinite(incScaled.sum())) incScaled.setZero();
        SE3 refToNew_new = SE3::exp((Vec6)(incScaled.head<6>()) * refToNew_current);
        AffLight aff_g2l_new = aff_g2l_current;
        aff_g2l_new.a += incScaled[6];
        aff_g2l_new.b += incScaled[7];
        Vec6 resNew = calcRes(lvl, refToNew_new, aff_g2l_new, setting_coarseCutoffTH*levelCutoffRepeat);
        bool accept = (resNew[0] / resNew[1]) < (resOld[0] / resOld[1]);
```

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/CoarseTracker.cpp#L536>

2.1. Frames → Pose Tracking

CoarseTracker.cpp::TrackNewestCoarse()::L536

```

int maxIterations[] = {10, 20, 50, 50, 50}; 각 피라미드 이미지 별 Iteration 횟수 설정
for(int lvl=coarsestLvl; lvl>=0; lvl--)
{
    Mat88 H; Vec8 b;
    float levelCutoffRepeat=1;
    Vec6 resOld = calcRes(lvl, refToNew_current, aff_g2l_current, setting_coarseCutoffTH*levelCutoffRepeat);
    while(resOld[5] > 0.6 && levelCutoffRepeat < 50)
    {
        levelCutoffRepeat*=2;
        resOld = calcRes(lvl, refToNew_current, aff_g2l_current, setting_coarseCutoffTH*levelCutoffRepeat);
        if(!setting_debugout_runquiet)
            printf("INCREASING cutoff to %f (ratio is %f)\n", setting_coarseCutoffTH*levelCutoffRepeat, resOld[5]);
    }
    calcGSSSE(lvl, H, b, refToNew_current, aff_g2l_current);
    float lambda = 0.01;
    for(int iteration=0; iteration < maxIterations[lvl]; iteration++)
    {
        Mat88 Hl = H;
        for(int i=0;i<8;i++) Hl(i,i) *= (1+lambda);
        Vec8 inc = Hl.ldlt().solve(-b);
        {...}
        float extrapFac = 1;
        if(lambda < lambdaExtrapolationLimit) extrapFac = sqrt(sqrt(lambdaExtrapolationLimit / lambda));
        inc *= extrapFac;
        Vec8 incScaled = inc;
        incScaled.segment<3>(0) *= SCALE_XI_ROT;
        incScaled.segment<3>(3) *= SCALE_XI_TRANS;
        incScaled.segment<1>(6) *= SCALE_A;
        incScaled.segment<1>(7) *= SCALE_B;
        if(!std::isfinite(incScaled.sum())) incScaled.setZero();
        SE3 refToNew_new = SE3::exp((Vec6)(incScaled.head<6>()) * refToNew_current);
        AffLight aff_g2l_new = aff_g2l_current;
        aff_g2l_new.a += incScaled[6];
        aff_g2l_new.b += incScaled[7];
        Vec6 resNew = calcRes(lvl, refToNew_new, aff_g2l_new, setting_coarseCutoffTH*levelCutoffRepeat);
        bool accept = (resNew[0] / resNew[1]) < (resOld[0] / resOld[1]);
    }
}

```

2.1. Frames → Pose Tracking

CODE REVIEW

CoarseTracker.cpp::TrackNewestCoarse()::L536

```
int maxIterations[] = {10, 20, 50, 50, 50};  
for(int lvl=coarsestLvl; lvl>=0; lvl--)  
{  
    Mat88 H; Vec8 b;  
    float levelCutoffRepeat=1;  
    Vec6 resOld = calcRes(lvl, refToNew_current, aff_g2l_current, setting_coarseCutoffTH*levelCutoffRepeat);  
    while(resOld[5] > 0.6 && levelCutoffRepeat < 50)  
    {  
        levelCutoffRepeat*=2;  
        resOld = calcRes(lvl, refToNew_current, aff_g2l_current, setting_coarseCutoffTH*levelCutoffRepeat);  
        if(!setting_debugout_runquiet)  
            printf("INCREASING cutoff to %f (ratio is %f)!\n", setting_coarseCutoffTH*levelCutoffRepeat, resOld[5]);  
    }  
    calcGSSSE(lvl, H, b, refToNew_current, aff_g2l_current);  
    float lambda = 0.01;  
    for(int iteration=0; iteration < maxIterations[lvl]; iteration++)  
    {  
        Mat88 Hl = H;  
        for(int i=0;i<8;i++) Hl(i,i) *= (1+lambda);  
        Vec8 inc = Hl.ldlt().solve(-b);  
        {...}  
        float extrapFac = 1;  
        if(lambda < lambdaExtrapolationLimit) extrapFac = sqrt(sqrt(lambdaExtrapolationLimit / lambda));  
        inc *= extrapFac;  
        Vec8 incScaled = inc;  
        incScaled.segment<3>(0) *= SCALE_XI_ROT;  
        incScaled.segment<3>(3) *= SCALE_XI_TRANS;  
        incScaled.segment<1>(6) *= SCALE_A;  
        incScaled.segment<1>(7) *= SCALE_B;  
        if(!std::isfinite(incScaled.sum())) incScaled.setZero();  
        SE3 refToNew_new = SE3::exp((Vec6)(incScaled.head<6>()) * refToNew_current;  
        AffLight aff_g2l_new = aff_g2l_current;  
        aff_g2l_new.a += incScaled[6];  
        aff_g2l_new.b += incScaled[7];  
        Vec6 resNew = calcRes(lvl, refToNew_new, aff_g2l_new, setting_coarseCutoffTH*levelCutoffRepeat);  
        bool accept = (resNew[0] / resNew[1]) < (resOld[0] / resOld[1]);
```

이미지 피라미드 별로 높은 피라미드(작은이미지)부터 시작

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/CoarseTracker.cpp#L536>

2.1. Frames → Pose Tracking

CODE REVIEW

CoarseTracker.cpp::TrackNewestCoarse()::L536

```
int maxIterations[] = {10, 20, 50, 50, 50};
for(int lvl=coarsestLvl; lvl>=0; lvl--)
{
    Mat88 H; Vec8 b;
    float levelCutoffRepeat=1;                                         최적화하기 전 residual 계산
    Vec6 resOld = calcRes(lvl, refToNew_current, aff_g2l_current, setting_coarseCutoffTH*levelCutoffRepeat);]
    while(resOld[5] > 0.6 && levelCutoffRepeat < 50)
    {
        levelCutoffRepeat*=2;
        resOld = calcRes(lvl, refToNew_current, aff_g2l_current, setting_coarseCutoffTH*levelCutoffRepeat);
        if(!setting_debugout_runquiet)
            printf("INCREASING cutoff to %f (ratio is %f)\n", setting_coarseCutoffTH*levelCutoffRepeat, resOld[5]);
    }
    calcGSSSE(lvl, H, b, refToNew_current, aff_g2l_current);
    float lambda = 0.01;
    for(int iteration=0; iteration < maxIterations[lvl]; iteration++)
    {
        Mat88 Hl = H;
        for(int i=0;i<8;i++) Hl(i,i) *= (1+lambda);
        Vec8 inc = Hl.ldlt().solve(-b);
        {...}
        float extrapFac = 1;
        if(lambda < lambdaExtrapolationLimit) extrapFac = sqrt(sqrt(lambdaExtrapolationLimit / lambda));
        inc *= extrapFac;
        Vec8 incScaled = inc;
        incScaled.segment<3>(0) *= SCALE_XI_ROT;
        incScaled.segment<3>(3) *= SCALE_XI_TRANS;
        incScaled.segment<1>(6) *= SCALE_A;
        incScaled.segment<1>(7) *= SCALE_B;
        if(!std::isfinite(incScaled.sum())) incScaled.setZero();
        SE3 refToNew_new = SE3::exp((Vec6)(incScaled.head<6>()) * refToNew_current;
        AffLight aff_g2l_new = aff_g2l_current;
        aff_g2l_new.a += incScaled[6];
        aff_g2l_new.b += incScaled[7];
        Vec6 resNew = calcRes(lvl, refToNew_new, aff_g2l_new, setting_coarseCutoffTH*levelCutoffRepeat);
        bool accept = (resNew[0] / resNew[1]) < (resOld[0] / resOld[1]);
```

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/CoarseTracker.cpp#L536>

2.1. Frames → Pose Tracking

CODE REVIEW

CoarseTracker.cpp::TrackNewestCoarse()::L536

```
int maxIterations[] = {10, 20, 50, 50, 50};
for(int lvl=coarsestLvl; lvl>=0; lvl--)
{
    Mat88 H; Vec8 b;
    float levelCutoffRepeat=1;
    Vec6 resOld = calcRes(lvl, refToNew_current, aff_g2l_current, setting_coarseCutoffTH*levelCutoffRepeat);
    while(resOld[5] > 0.6 && levelCutoffRepeat < 50)
    {
        levelCutoffRepeat*=2;
        resOld = calcRes(lvl, refToNew_current, aff_g2l_current, setting_coarseCutoffTH*levelCutoffRepeat);
        if(!setting_debugout_runquiet)
            printf("INCREASING cutoff to %f (ratio is %f)!\n", setting_coarseCutoffTH*levelCutoffRepeat, resOld[5]);
    }
    calcGSSSE(lvl, H, b, refToNew_current, aff_g2l_current); H, b 를 계산한다
    float lambda = 0.01;
    for(int iteration=0; iteration < maxIterations[lvl]; iteration++)
    {
        Mat88 Hl = H;
        for(int i=0;i<8;i++) Hl(i,i) *= (1+lambda);
        Vec8 inc = Hl.ldlt().solve(-b);
        {...}
        float extrapFac = 1;
        if(lambda < lambdaExtrapolationLimit) extrapFac = sqrt(sqrt(lambdaExtrapolationLimit / lambda));
        inc *= extrapFac;
        Vec8 incScaled = inc;
        incScaled.segment<3>(0) *= SCALE_XI_ROT;
        incScaled.segment<3>(3) *= SCALE_XI_TRANS;
        incScaled.segment<1>(6) *= SCALE_A;
        incScaled.segment<1>(7) *= SCALE_B;
        if(!std::isfinite(incScaled.sum())) incScaled.setZero();
        SE3 refToNew_new = SE3::exp((Vec6)(incScaled.head<6>()) * refToNew_current;
        AffLight aff_g2l_new = aff_g2l_current;
        aff_g2l_new.a += incScaled[6];
        aff_g2l_new.b += incScaled[7];
        Vec6 resNew = calcRes(lvl, refToNew_new, aff_g2l_new, setting_coarseCutoffTH*levelCutoffRepeat);
        bool accept = (resNew[0] / resNew[1]) < (resOld[0] / resOld[1]);
```

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/CoarseTracker.cpp#L536>

2.1. Frames → Pose Tracking

CODE REVIEW

CoarseTracker.cpp::TrackNewestCoarse()::L536

```
int maxIterations[] = {10, 20, 50, 50, 50};
for(int lvl=coarsestLvl; lvl>=0; lvl--)
{
    Mat88 H; Vec8 b;
    float levelCutoffRepeat=1;
    Vec6 resOld = calcRes(lvl, refToNew_current, aff_g2l_current, setting_coarseCutoffTH*levelCutoffRepeat);
    while(resOld[5] > 0.6 && levelCutoffRepeat < 50)
    {
        levelCutoffRepeat*=2;
        resOld = calcRes(lvl, refToNew_current, aff_g2l_current, setting_coarseCutoffTH*levelCutoffRepeat);
        if(!setting_debugout_runquiet)
            printf("INCREASING cutoff to %f (ratio is %f)\n", setting_coarseCutoffTH*levelCutoffRepeat, resOld[5]);
    }
    calcGSSSE(lvl, H, b, refToNew_current, aff_g2l_current);
    float lambda = 0.01; 초기 lambda 값 설정
    for(int iteration=0; iteration < maxIterations[lvl]; iteration++)
    {
        Mat88 Hl = H;
        for(int i=0;i<8;i++) Hl(i,i) *= (1+lambda);
        Vec8 inc = Hl.ldlt().solve(-b);
        {...}
        float extrapFac = 1;
        if(lambda < lambdaExtrapolationLimit) extrapFac = sqrt(sqrt(lambdaExtrapolationLimit / lambda));
        inc *= extrapFac;
        Vec8 incScaled = inc;
        incScaled.segment<3>(0) *= SCALE_XI_ROT;
        incScaled.segment<3>(3) *= SCALE_XI_TRANS;
        incScaled.segment<1>(6) *= SCALE_A;
        incScaled.segment<1>(7) *= SCALE_B;
        if(!std::isfinite(incScaled.sum())) incScaled.setZero();
        SE3 refToNew_new = SE3::exp((Vec6)(incScaled.head<6>()) * refToNew_current);
        AffLight aff_g2l_new = aff_g2l_current;
        aff_g2l_new.a += incScaled[6];
        aff_g2l_new.b += incScaled[7];
        Vec6 resNew = calcRes(lvl, refToNew_new, aff_g2l_new, setting_coarseCutoffTH*levelCutoffRepeat);
        bool accept = (resNew[0] / resNew[1]) < (resOld[0] / resOld[1]);
```

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/CoarseTracker.cpp#L536>

2.1. Frames → Pose Tracking

CODE REVIEW

CoarseTracker.cpp::TrackNewestCoarse()::L536

```
int maxIterations[] = {10, 20, 50, 50, 50};
for(int lvl=coarsestLvl; lvl>=0; lvl--)
{
    Mat88 H; Vec8 b;
    float levelCutoffRepeat=1;
    Vec6 resOld = calcRes(lvl, refToNew_current, aff_g2l_current, setting_coarseCutoffTH*levelCutoffRepeat);
    while(resOld[5] > 0.6 && levelCutoffRepeat < 50)
    {
        levelCutoffRepeat*=2;
        resOld = calcRes(lvl, refToNew_current, aff_g2l_current, setting_coarseCutoffTH*levelCutoffRepeat);
        if(!setting_debugout_runquiet)
            printf("INCREASING cutoff to %f (ratio is %f)!\n", setting_coarseCutoffTH*levelCutoffRepeat, resOld[5]);
    }
    calcGSSSE(lvl, H, b, refToNew_current, aff_g2l_current);
    float lambda = 0.01;
    for(int iteration=0; iteration < maxIterations[lvl]; iteration++)
    {
        Mat88 Hl = H;
        for(int i=0;i<8;i++) Hl(i,i) *= (1+lambda);
        Vec8 inc = Hl.ldlt().solve(-b);
        {...}
        float extrapFac = 1;
        if(lambda < lambdaExtrapolationLimit) extrapFac = sqrt(sqrt(lambdaExtrapolationLimit / lambda));
        inc *= extrapFac;
        Vec8 incScaled = inc;
        incScaled.segment<3>(0) *= SCALE_XI_ROT;
        incScaled.segment<3>(3) *= SCALE_XI_TRANS;
        incScaled.segment<1>(6) *= SCALE_A;
        incScaled.segment<1>(7) *= SCALE_B;
        if(!std::isfinite(incScaled.sum())) incScaled.setZero();
        SE3 refToNew_new = SE3::exp((Vec6)(incScaled.head<6>()) * refToNew_current);
        AffLight aff_g2l_new = aff_g2l_current;
        aff_g2l_new.a += incScaled[6];
        aff_g2l_new.b += incScaled[7];
        Vec6 resNew = calcRes(lvl, refToNew_new, aff_g2l_new, setting_coarseCutoffTH*levelCutoffRepeat);
        bool accept = (resNew[0] / resNew[1]) < (resOld[0] / resOld[1]);
```

LM 방법을 통한 최적화

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/CoarseTracker.cpp#L536>

2.1. Frames → Pose Tracking

CoarseTracker.cpp::TrackNewestCoarse()::L536

```

int maxIterations[] = {10, 20, 50, 50, 50};
for(int lvl=coarsestLvl; lvl>=0; lvl--)
{
    Mat88 H; Vec8 b;
    float levelCutoffRepeat=1;
    Vec6 resOld = calcRes(lvl, refToNew_current, aff_g2l_current, setting_coarseCutoffTH*levelCutoffRepeat);
    while(resOld[5] > 0.6 && levelCutoffRepeat < 50)
    {
        levelCutoffRepeat*=2;
        resOld = calcRes(lvl, refToNew_current, aff_g2l_current, setting_coarseCutoffTH*levelCutoffRepeat);
        if(!setting_debugout_runquiet)
            printf("INCREASING cutoff to %f (ratio is %f)!\n", setting_coarseCutoffTH*levelCutoffRepeat, resOld[5]);
    }
    calcGSSSE(lvl, H, b, refToNew_current, aff_g2l_current);
    float lambda = 0.01;
    for(int iteration=0; iteration < maxIterations[lvl]; iteration++)
    {
        Mat88 Hl = H;
        for(int i=0;i<8;i++) Hl(i,i) *= (1+lambda);
        Vec8 inc = Hl.Ildt().solve(-b);
        ...
        float extrapFac = 1;
        if(lambda < lambdaExtrapolationLimit) extrapFac = sqrt(sqrt(lambdaExtrapolationLimit / lambda));
        inc *= extrapFac;
        Vec8 incScaled = inc;
        incScaled.segment<3>(0) *= SCALE_XI_ROT;
        incScaled.segment<3>(3) *= SCALE_XI_TRANS;
        incScaled.segment<1>(6) *= SCALE_A;
        incScaled.segment<1>(7) *= SCALE_B;
        if(!std::isfinite(incScaled.sum())) incScaled.setZero();
        SE3 refToNew_new = SE3::exp((Vec6)(incScaled.head<6>()) * refToNew_current);
        AffLight aff_g2l_new = aff_g2l_current;
        aff_g2l_new.a += incScaled[6];
        aff_g2l_new.b += incScaled[7];
        Vec6 resNew = calcRes(lvl, refToNew_new, aff_g2l_new, setting_coarseCutoffTH*levelCutoffRepeat);
        bool accept = (resNew[0] / resNew[1]) < (resOld[0] / resOld[1]);
    }
}

```

pose 증분량 계산

$$\begin{aligned} (\mathbf{H} + \lambda\mathbf{I})\Delta\mathbf{x} &= \mathbf{b} \\ \Delta\mathbf{x} &= (\mathbf{H} + \lambda\mathbf{I})^{-1}\mathbf{b} \end{aligned}$$

2.1. Frames → Pose Tracking

CoarseTracker.cpp::TrackNewestCoarse()::L536

```

int maxIterations[] = {10, 20, 50, 50, 50};
for(int lvl=coarsestLvl; lvl>=0; lvl--)
{
    Mat88 H; Vec8 b;
    float levelCutoffRepeat=1;
    Vec6 resOld = calcRes(lvl, refToNew_current, aff_g21_current, setting_coarseCutoffTH*levelCutoffRepeat);
    while(resOld[5] > 0.6 && levelCutoffRepeat < 50)
    {
        levelCutoffRepeat*=2;
        resOld = calcRes(lvl, refToNew_current, aff_g21_current, setting_coarseCutoffTH*levelCutoffRepeat);
        if(!setting_debugout_runquiet)
            printf("INCREASING cutoff to %f (ratio is %f)!\n", setting_coarseCutoffTH*levelCutoffRepeat, resOld[5]);
    }
    calcGSSSE(lvl, H, b, refToNew_current, aff_g21_current);
    float lambda = 0.01;
    for(int iteration=0; iteration < maxIterations[lvl]; iteration++)
    {
        Mat88 Hl = H;
        for(int i=0;i<8;i++) Hl(i,i) *= (1+lambda);
        Vec8 inc = Hl.ldlt().solve(-b);
        {...}
        float extrapFac = 1;
        if(lambda < lambdaExtrapolationLimit) extrapFac = sqrt(sqrt(lambdaExtrapolationLimit / lambda));
        inc *= extrapFac;
        Vec8 incScaled = inc;
        incScaled.segment<3>(0) *= SCALE_XI_ROT;
        incScaled.segment<3>(3) *= SCALE_XI_TRANS;
        incScaled.segment<1>(6) *= SCALE_A;
        incScaled.segment<1>(7) *= SCALE_B;
        if(!std::isfinite(incScaled.sum())) incScaled.setZero();
        SE3 refToNew_new = SE3::exp((Vec6)(incScaled.head<6>()) * refToNew_current);
        AffLight aff_g21_new = aff_g21_current;
        aff_g21_new.a += incScaled[6];
        aff_g21_new.b += incScaled[7];
        Vec6 resNew = calcRes(lvl, refToNew_new, aff_g21_new, setting_coarseCutoffTH*levelCutoffRepeat);
        bool accept = (resNew[0] / resNew[1]) < (resOld[0] / resOld[1]);
    }
}

```

pose 업데이트

$$\mathbf{x} \leftarrow \exp(\Delta\mathbf{x}^\wedge)\mathbf{x}$$

2.1. Frames → Pose Tracking

CODE REVIEW

CoarseTracker.cpp::TrackNewestCoarse()::L536

```
int maxIterations[] = {10, 20, 50, 50, 50};
for(int lvl=coarsestLvl; lvl>=0; lvl--)
{
    Mat88 H; Vec8 b;
    float levelCutoffRepeat=1;
    Vec6 resOld = calcRes(lvl, refToNew_current, aff_g2l_current, setting_coarseCutoffTH*levelCutoffRepeat);
    while(resOld[5] > 0.6 && levelCutoffRepeat < 50)
    {
        levelCutoffRepeat*=2;
        resOld = calcRes(lvl, refToNew_current, aff_g2l_current, setting_coarseCutoffTH*levelCutoffRepeat);
        if(!setting_debugout_runquiet)
            printf("INCREASING cutoff to %f (ratio is %f)\n", setting_coarseCutoffTH*levelCutoffRepeat, resOld[5]);
    }
    calcGSSSE(lvl, H, b, refToNew_current, aff_g2l_current);
    float lambda = 0.01;
    for(int iteration=0; iteration < maxIterations[lvl]; iteration++)
    {
        Mat88 Hl = H;
        for(int i=0;i<8;i++) Hl(i,i) *= (1+lambda);
        Vec8 inc = Hl.ldlt().solve(-b);
        {...}
        float extrapFac = 1;
        if(lambda < lambdaExtrapolationLimit) extrapFac = sqrt(sqrt(lambdaExtrapolationLimit / lambda));
        inc *= extrapFac;
        Vec8 incScaled = inc;
        incScaled.segment<3>(0) *= SCALE_XI_ROT;
        incScaled.segment<3>(3) *= SCALE_XI_TRANS;
        incScaled.segment<1>(6) *= SCALE_A;
        incScaled.segment<1>(7) *= SCALE_B;
        if(!std::isfinite(incScaled.sum())) incScaled.setZero();
        SE3 refToNew_new = SE3::exp((Vec6)(incScaled.head<6>()) * refToNew_current);
        AffLight aff_g2l_new = aff_g2l_current;
        aff_g2l_new.a += incScaled[6];
        aff_g2l_new.b += incScaled[7];
        Vec6 resNew = calcRes(lvl, refToNew_new, aff_g2l_new, setting_coarseCutoffTH*levelCutoffRepeat);
        bool accept = (resNew[0] / resNew[1]) < (resOld[0] / resOld[1]);
    }
}
```

pose 업데이트 후 다시 residual을 계산해서 residual이 감소하였는지 확인
감소하였으면 해당 pose 업데이트 적용.
감소하지 않았으면 다른 후보 pose 선택.

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/CoarseTracker.cpp#L536>

3.1. Non-Keyframes → Inverse Depth Update

FullSystem.cpp::addActiveFrame():L881

```
// BRIGHTNESS CHECK
needToMakeKF = allFrameHistory.size() == 1 ||
    setting_kfGlobalWeight*setting_maxShiftWeightT * sqrtf((double)tres[1]) / (wG[0]+hG[0]) +
    setting_kfGlobalWeight*setting_maxShiftWeightR * sqrtf((double)tres[2]) / (wG[0]+hG[0]) +
    setting_kfGlobalWeight*setting_maxShiftWeightRT * sqrtf((double)tres[3]) / (wG[0]+hG[0]) +
    setting_kfGlobalWeight*setting_maxAffineWeight * fabs(logf((float)refToFh[0])) > 1 ||
    2*coarseTracker->firstCoarseRMSE < tres[0];
```

FullSystem.cpp::makeNonKeyFrame():L1027

```
void FullSystem::makeNonKeyFrame( FrameHessian* fh)
{
    // needs to be set by mapping thread. no lock required since we are in mapping thread.
    {
        boost::unique_lock<boost::mutex> crlock(shellPoseMutex);
        assert(fh->shell->trackingRef != 0);
        fh->shell->camToWorld = fh->shell->trackingRef->camToWorld * fh->shell->camToTrackingRef;
        fh->setEvalPT_scaled(fh->shell->camToWorld.inverse(),fh->shell->aff_g2l);
    }
    traceNewCoarse(fh);
    delete fh;
}
```

FullSystem.cpp::traceNewCoarse():L463

```
void FullSystem::traceNewCoarse(FrameHessian* fh) {
    boost::unique_lock<boost::mutex> lock(mapMutex);
    int trace_total=0, trace_good=0, trace_oob=0, trace_out=0, trace_skip=0, trace_badcondition=0, trace_uninitialized=0;
    Mat33f K = Mat33f::Identity();
    K(0,0) = Hcalib.fx1();
    K(1,1) = Hcalib.fy1();
    K(0,2) = Hcalib.cx1();
    K(1,2) = Hcalib.cy1();
    for(FrameHessian* host : frameHessians) { // go through all active frames
        SE3 hostToNew = fh->PRE_worldToCam * host->PRE_camToWorld;
        Mat33f KRKi = K * hostToNew.rotationMatrix().cast<float>() * K.inverse();
        Vec3f Kt = K * hostToNew.translation().cast<float>();
        Vec2f aff = AfflFlight::fromToVecExposure(host->ab_exposure, fh->ab_exposure, host->aff_g2l(), fh->aff_g2l()).cast<float>();
        for(ImmaturePoint* ph : host->immaturePoints)
        {
            ph->traceOn(fh, KRKi, Kt, aff, &Hcalib, false );
        }
    }
}
```

3.1. Non-Keyframes → Inverse Depth Update

FullSystem.cpp::addActiveFrame():L881

현재 프레임을 키프레임으로 선정하기 위한 조건

```
// BRIGHTNESS CHECK
needToMakeKF = allFrameHistory.size() == 1 ||
    setting_kfGlobalWeight*setting_maxShiftWeightT * sqrtf((double)tres[1]) / (wG[0]+hG[0]) +
    setting_kfGlobalWeight*setting_maxShiftWeightR * sqrtf((double)tres[2]) / (wG[0]+hG[0]) +
    setting_kfGlobalWeight*setting_maxShiftWeightRT * sqrtf((double)tres[3]) / (wG[0]+hG[0]) +
    setting_kfGlobalWeight*setting_maxAffineWeight * fabs(logf((float)refToFh[0])) > 1 ||
    2*coarseTracker->firstCoarseRMSE < tres[0];
```

1. 해당 프레임이 가장 첫 번째 프레임인가?
2. (순수 회전을 제외한) 카메라 움직임으로 인한 Optical flow의 변화량 + Photometric Params(a,b)의 변화량이 일정 기준 이상인가?
3. Residual 값이 이전 키프레임과 비교했을 때 급격하게 변했는가?

FullSystem.cpp::makeNonKeyFrame():L1027

```
void FullSystem::makeNonKeyFrame( FrameHessian* fh)
{
    // needs to be set by mapping thread. no lock required since we are in mapping thread.
    {
        boost::unique_lock<boost::mutex> crlock(shellPoseMutex);
        assert(fh->shell->trackingRef != 0);
        fh->shell->camToWorld = fh->shell->trackingRef->camToWorld * fh->shell->camToTrackingRef;
        fh->setEvalPT_scaled(fh->shell->camToWorld.inverse(), fh->shell->aff_g2l);
    }
    traceNewCoarse(fh);
    delete fh;
}
```

FullSystem.cpp::traceNewCoarse():L463

```
void FullSystem::traceNewCoarse(FrameHessian* fh) {
    boost::unique_lock<boost::mutex> lock(mapMutex);
    int trace_total=0, trace_good=0, trace_oob=0, trace_out=0, trace_skip=0, trace_badcondition=0, trace_uninitialized=0;
    Mat33f K = Mat33f::Identity();
    K(0,0) = Hcalib.fx1();
    K(1,1) = Hcalib.fy1();
    K(0,2) = Hcalib.cx1();
    K(1,2) = Hcalib.cy1();
    for(FrameHessian* host : frameHessians) { // go through all active frames
        SE3 hostToWorld = fh->PRE_worldToCam * host->PRE_camToWorld;
        Mat33f KRKi = K * hostToWorld.rotationMatrix().cast<float>() * K.inverse();
        Vec3f Kt = K * hostToWorld.translation().cast<float>();
        Vec2f aff = AfflFlight::fromToVecExposure(host->ab_exposure, fh->ab_exposure, host->aff_g2l(), fh->aff_g2l()).cast<float>();
        for(ImmaturePoint* ph : host->immaturePoints)
        {
            ph->traceOn(fh, KRKi, Kt, aff, &Hcalib, false );
        }
    }
}
```

3.1. Non-Keyframes → Inverse Depth Update

FullSystem.cpp::addActiveFrame():L881

```
// BRIGHTNESS CHECK
needToMakeKF = allFrameHistory.size() == 1 ||
    setting_kfGlobalWeight*setting_maxShiftWeightT * sqrtf((double)tres[1]) / (wG[0]+hG[0]) +
    setting_kfGlobalWeight*setting_maxShiftWeightR * sqrtf((double)tres[2]) / (wG[0]+hG[0]) +
    setting_kfGlobalWeight*setting_maxShiftWeightRT * sqrtf((double)tres[3]) / (wG[0]+hG[0]) +
    setting_kfGlobalWeight*setting_maxAffineWeight * fabs(logf((float)refToFh[0])) > 1 ||
    2*coarseTracker->firstCoarseRMSE < tres[0];
```

FullSystem.cpp::makeNonKeyFrame():L1027

```
void FullSystem::makeNonKeyFrame( FrameHessian* fh)
{
    // needs to be set by mapping thread. no lock required since we are in mapping thread.
    {
        boost::unique_lock<boost::mutex> crlock(shellPoseMutex);
        assert(fh->shell->trackingRef != 0);
        fh->shell->camToWorld = fh->shell->trackingRef->camToWorld * fh->shell->camToTrackingRef;
        fh->setEvalPT_scaled(fh->shell->camToWorld.inverse(),fh->shell->aff_g2l);
    }
    traceNewCoarse(fh);
    delete fh;
}
```

키프레임으로 선정되지 않은 경우 실행되는 함수
traceNewCoarse() 함수 호출

FullSystem.cpp::traceNewCoarse():L463

```
void FullSystem::traceNewCoarse(FrameHessian* fh) {
    boost::unique_lock<boost::mutex> lock(mapMutex);
    int trace_total=0, trace_good=0, trace_oob=0, trace_out=0, trace_skip=0, trace_badcondition=0, trace_uninitialized=0;
    Mat33f K = Mat33f::Identity();
    K(0,0) = Hcalib.fx1();
    K(1,1) = Hcalib.fy1();
    K(0,2) = Hcalib.cx1();
    K(1,2) = Hcalib.cy1();
    for(FrameHessian* host : frameHessians) { // go through all active frames
        SE3 hostToNew = fh->PRE_worldToCam * host->PRE_camToWorld;
        Mat33f KRKi = K * hostToNew.rotationMatrix().cast<float>() * K.inverse();
        Vec3f Kt = K * hostToNew.translation().cast<float>();
        Vec2f aff = AfflFlight::fromToVecExposure(host->ab_exposure, fh->ab_exposure, host->aff_g2l(), fh->aff_g2l()).cast<float>();
        for(ImmaturePoint* ph : host->immaturePoints)
        {
            ph->traceOn(fh, KRKi, Kt, aff, &Hcalib, false );
        }
    }
}
```

3.1. Non-Keyframes → Inverse Depth Update

FullSystem.cpp::addActiveFrame():L881

```
// BRIGHTNESS CHECK
needToMakeKF = allFrameHistory.size() == 1 ||
    setting_kfGlobalWeight*setting_maxShiftWeightT * sqrtf((double)tres[1]) / (wG[0]+hG[0]) +
    setting_kfGlobalWeight*setting_maxShiftWeightR * sqrtf((double)tres[2]) / (wG[0]+hG[0]) +
    setting_kfGlobalWeight*setting_maxShiftWeightRT * sqrtf((double)tres[3]) / (wG[0]+hG[0]) +
    setting_kfGlobalWeight*setting_maxAffineWeight * fabs(logf((float)refToFh[0])) > 1 ||
    2*coarseTracker->firstCoarseRMSE < tres[0];
```

FullSystem.cpp::makeNonKeyFrame():L1027

```
void FullSystem::makeNonKeyFrame( FrameHessian* fh)
{
    // needs to be set by mapping thread. no lock required since we are in mapping thread.
    {
        boost::unique_lock<boost::mutex> crlock(shellPoseMutex);
        assert(fh->shell->trackingRef != 0);
        fh->shell->camToWorld = fh->shell->trackingRef->camToWorld * fh->shell->camToTrackingRef;
        fh->setEvalPT_scaled(fh->shell->camToWorld.inverse(),fh->shell->aff_g2l);
    }
    traceNewCoarse(fh);
    delete fh;
}
```

FullSystem.cpp::traceNewCoarse():L463

```
void FullSystem::traceNewCoarse(FrameHessian* fh) {
    boost::unique_lock<boost::mutex> lock(mapMutex);
    int trace_total=0, trace_good=0, trace_oob=0, trace_out=0, trace_skip=0, trace_badcondition=0, trace_uninitialized=0;
    Mat33f K = Mat33f::Identity();
    K(0,0) = Hcalib.fx1();
    K(1,1) = Hcalib.fy1();          카메라 내부 파라미터 설정
    K(0,2) = Hcalib.cx1();
    K(1,2) = Hcalib.cy1();
    for(FrameHessian* host : frameHessians) { // go through all active frames
        SE3 hostToNew = fh->PRE_worldToCam * host->PRE_camToWorld;
        Mat33f KRKi = K * hostToNew.rotationMatrix().cast<float>() * K.inverse();
        Vec3f Kt = K * hostToNew.translation().cast<float>();
        Vec2f aff = AffLight::fromToVecExposure(host->ab_exposure, fh->ab_exposure, host->aff_g2l(), fh->aff_g2l()).cast<float>();
        for(ImmaturePoint* ph : host->immaturePoints)
        {
            ph->traceOn(fh, KRKi, Kt, aff, &Hcalib, false );
        }
    }
}
```

3.1. Non-Keyframes → Inverse Depth Update

FullSystem.cpp::addActiveFrame():L881

```
// BRIGHTNESS CHECK
needToMakeKF = allFrameHistory.size() == 1 ||
    setting_kfGlobalWeight*setting_maxShiftWeightT * sqrtf((double)tres[1]) / (wG[0]+hG[0]) +
    setting_kfGlobalWeight*setting_maxShiftWeightR * sqrtf((double)tres[2]) / (wG[0]+hG[0]) +
    setting_kfGlobalWeight*setting_maxShiftWeightRT * sqrtf((double)tres[3]) / (wG[0]+hG[0]) +
    setting_kfGlobalWeight*setting_maxAffineWeight * fabs(logf((float)refToFh[0])) > 1 ||
    2*coarseTracker->firstCoarseRMSE < tres[0];
```

FullSystem.cpp::makeNonKeyFrame():L1027

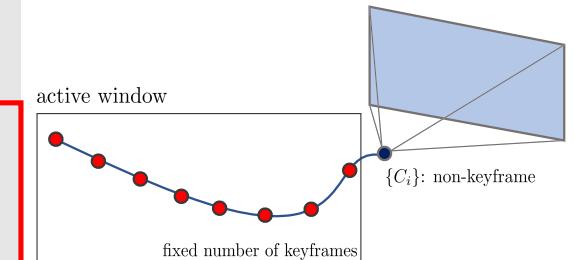
```
void FullSystem::makeNonKeyFrame( FrameHessian* fh)
{
    // needs to be set by mapping thread. no lock required since we are in mapping thread.
    {
        boost::unique_lock<boost::mutex> crlock(shellPoseMutex);
        assert(fh->shell->trackingRef != 0);
        fh->shell->camToWorld = fh->shell->trackingRef->camToWorld * fh->shell->camToTrackingRef;
        fh->setEvalPT_scaled(fh->shell->camToWorld.inverse(),fh->shell->aff_g2l);
    }
    traceNewCoarse(fh);
    delete fh;
}
```

FullSystem.cpp::traceNewCoarse():L463

```
void FullSystem::traceNewCoarse(FrameHessian* fh) {
    boost::unique_lock<boost::mutex> lock(mapMutex);
    int trace_total=0, trace_good=0, trace_oob=0, trace_out=0, trace_skip=0, trace_badcondition=0, trace_uninitialized=0;
    Mat33f K = Mat33f::Identity();
    K(0,0) = Hcalib.fx1();
    K(1,1) = Hcalib.fy1();
    K(0,2) = Hcalib.cxl();
    K(1,2) = Hcalib.cyl();
```

active window (frameHessians)에 있는 모든 키프레임에 대해 루프를 돌면서

```
for(FrameHessian* host : frameHessians) { // go through all active frames
    SE3 hostToNew = fh->PRE_worldToCam * host->PRE_camToWorld;
    Mat33f KRKi = K * hostToNew.rotationMatrix().cast<float>() * K.inverse();
    Vec3f Kt = K * hostToNew.translation().cast<float>();
    Vec2f aff = AfflFlight::fromToVecExposure(host->ab_exposure, fh->ab_exposure, host->aff_g2l(), fh->aff_g2l()).cast<float>();
    for(ImmaturePoint* ph : host->immaturePoints)
    {
        ph->traceOn(fh, KRKi, Kt, aff, &Hcalib, false );
    }
}
```



3.1. Non-Keyframes → Inverse Depth Update

FullSystem.cpp::addActiveFrame():L881

```
// BRIGHTNESS CHECK
needToMakeKF = allFrameHistory.size() == 1 ||
    setting_kfGlobalWeight*setting_maxShiftWeightT * sqrtf((double)tres[1]) / (wG[0]+hG[0]) +
    setting_kfGlobalWeight*setting_maxShiftWeightR * sqrtf((double)tres[2]) / (wG[0]+hG[0]) +
    setting_kfGlobalWeight*setting_maxShiftWeightRT * sqrtf((double)tres[3]) / (wG[0]+hG[0]) +
    setting_kfGlobalWeight*setting_maxAffineWeight * fabs(logf((float)refToFh[0])) > 1 ||
    2*coarseTracker->firstCoarseRMSE < tres[0];
```

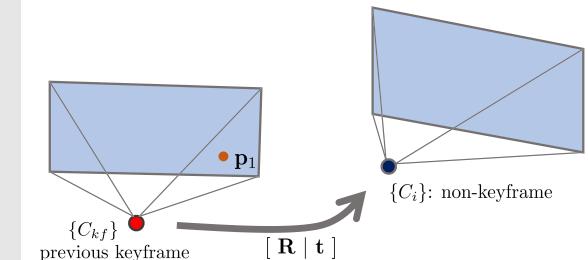
FullSystem.cpp::makeNonKeyFrame():L1027

```
void FullSystem::makeNonKeyFrame( FrameHessian* fh)
{
    // needs to be set by mapping thread. no lock required since we are in mapping thread.
    {
        boost::unique_lock<boost::mutex> crlock(shellPoseMutex);
        assert(fh->shell->trackingRef != 0);
        fh->shell->camToWorld = fh->shell->trackingRef->camToWorld * fh->shell->camToTrackingRef;
        fh->setEvalPT_scaled(fh->shell->camToWorld.inverse(), fh->shell->aff_g2l);
    }
    traceNewCoarse(fh);
    delete fh;
}
```

FullSystem.cpp::traceNewCoarse():L463

```
void FullSystem::traceNewCoarse(FrameHessian* fh) {
    boost::unique_lock<boost::mutex> lock(mapMutex);
    int trace_total=0, trace_good=0, trace_oob=0, trace_out=0, trace_skip=0, trace_badcondition=0, trace_uninitialized=0;
    Mat33f K = Mat33f::Identity();
    K(0,0) = Hcalib.fx1();
    K(1,1) = Hcalib.fy1();
    K(0,2) = Hcalib.cxl();
    K(1,2) = Hcalib.cyl();
    for(FrameHessian* host : frameHessians) { // go through all active frames
        SE3 hostToNew = fh->PRE_worldToCam * host->PRE_camToWorld;
        Mat33f KRKi = K * hostToNew.rotationMatrix().cast<float>() * K.inverse();
        Vec3f Kt = K * hostToNew.translation().cast<float>();
        Vec2f aff = AffLight::fromToVecExposure(host->ab_exposure, fh->ab_exposure, host->aff_g2l(), fh->aff_g2l()).cast<float>();
        for(ImmaturePoint* ph : host->immaturePoints)
        {
            ph->traceOn(fh, KRKi, Kt, aff, &Hcalib, false );
        }
    }
}
```

모든 키프레임에 immaturePoint들을 현재 프레임에 프로젝션시킨 후 idepth를 업데이트하는 traceOn() 함수 실행



3.1. Non-Keyframes → Inverse Depth Update

ImmaturePoint.cpp::traceOn()::L97

```

Vec3f pr = hostToFrame_KRKi * Vec3f(u,v, 1);
Vec3f ptpMin = pr + hostToFrame_Kt*idepth_min;
float uMin = ptpMin[0] / ptpMin[2];
float vMin = ptpMin[1] / ptpMin[2];
{...}
float dist;
float uMax;
float vMax;
Vec3f ptpMax;

if(std::isfinite(idepth_max))
{
    ptpMax = pr + hostToFrame_Kt*idepth_max;
    uMax = ptpMax[0] / ptpMax[2];
    vMax = ptpMax[1] / ptpMax[2];
    {...}
}
else
{
    dist = maxPixSearch;
    // project to arbitrary depth to get direction.
    ptpMax = pr + hostToFrame_Kt*0.01;
    uMax = ptpMax[0] / ptpMax[2];
    vMax = ptpMax[1] / ptpMax[2];
    // direction.
    float dx = uMax-uMin;
    float dy = vMax-vMin;
    float d = 1.0f / sqrtf(dx*dx+dy*dy);
    // set to [setting_maxPixSearch].
    uMax = uMin + dist*dx*d;
    vMax = vMin + dist*dy*d;
}

```

3.1. Non-Keyframes → Inverse Depth Update

ImmaturePoint.cpp::traceOn()::L97

```

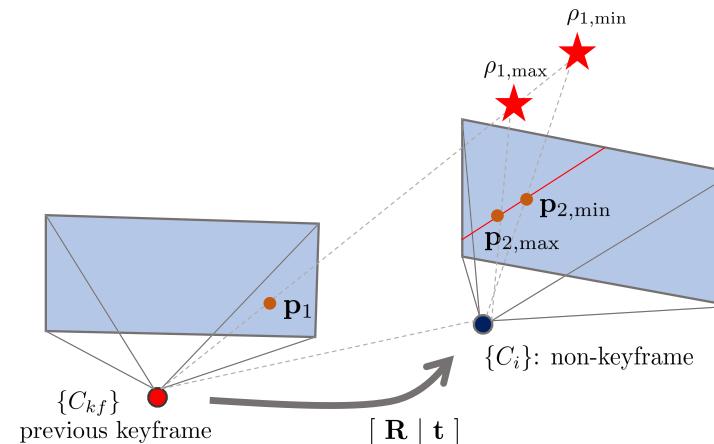
Vec3f pr = hostToFrame_KRKi * Vec3f(u,v, 1);
Vec3f ptpMin = pr + hostToFrame_Kt*idepth_min;
float uMin = ptpMin[0] / ptpMin[2];
float vMin = ptpMin[1] / ptpMin[2];
...
float dist;
float uMax;
float vMax;
Vec3f ptpMax;

if(std::isfinite(idepth_max))
{
    ptpMax = pr + hostToFrame_Kt*idepth_max;
    uMax = ptpMax[0] / ptpMax[2];
    vMax = ptpMax[1] / ptpMax[2];
    ...
}
else
{
    dist = maxPixSearch;
    // project to arbitrary depth to get direction.
    ptpMax = pr + hostToFrame_Kt*0.01;
    uMax = ptpMax[0] / ptpMax[2];
    vMax = ptpMax[1] / ptpMax[2];
    // direction.
    float dx = uMax-uMin;
    float dy = vMax-vMin;
    float d = 1.0f / sqrtf(dx*dx+dy*dy);
    // set to [setting_maxPixSearch].
    uMax = uMin + dist*dx*d;
    vMax = vMin + dist*dy*d;
}

```

$$\mathbf{p}_{2r} = \mathbf{K} \mathbf{R} \mathbf{K}^{-1} \mathbf{p}_1$$

$$\mathbf{p}_{2,min} = \mathbf{p}_{2r} + \mathbf{K} \mathbf{t} \rho_{1,min}$$



★ 3D point
— epipolar line

$$\mathbf{p}_{2r} = \mathbf{K} \mathbf{R} \mathbf{K}^{-1} \mathbf{p}_1$$

$$\mathbf{p}_{2,min} = \mathbf{p}_{2r} + \mathbf{K} \mathbf{t} \rho_{1,min}$$

$$\mathbf{p}_{2,max} = \mathbf{p}_{2r} + \mathbf{K} \mathbf{t} \rho_{1,max}$$

3.1. Non-Keyframes → Inverse Depth Update

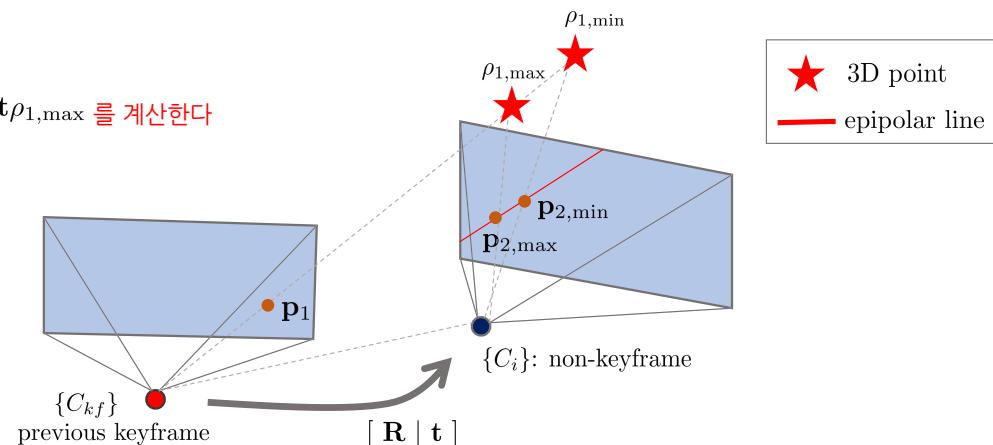
ImmaturePoint.cpp::traceOn()::L97

```

Vec3f pr = hostToFrame_KRKi * Vec3f(u,v, 1);
Vec3f ptpMin = pr + hostToFrame_Kt*idepth_min;
float uMin = ptpMin[0] / ptpMin[2];
float vMin = ptpMin[1] / ptpMin[2];
{...}
float dist;
float uMax;
float vMax;
Vec3f ptpMax;    idepth_max 값이 있는 경우와 없는 경우로 나눠서  $\mathbf{p}_{2,max} = \mathbf{p}_{2r} + \mathbf{Kt}\rho_{1,max}$  를 계산한다

if(std::isfinite(idepth_max))
{
    ptpMax = pr + hostToFrame_Kt*idepth_max;
    uMax = ptpMax[0] / ptpMax[2];
    vMax = ptpMax[1] / ptpMax[2];
    {...}
}
else
{
    dist = maxPixSearch;
    // project to arbitrary depth to get direction.
    ptpMax = pr + hostToFrame_Kt*0.01;
    uMax = ptpMax[0] / ptpMax[2];
    vMax = ptpMax[1] / ptpMax[2];
    // direction.
    float dx = uMax-uMin;
    float dy = vMax-vMin;
    float d = 1.0f / sqrtf(dx*dx+dy*dy);
    // set to [setting_maxPixSearch].
    uMax = uMin + dist*dx*d;
    vMax = vMin + dist*dy*d;
}

```



$$\begin{aligned}\mathbf{p}_{2r} &= \mathbf{K} \mathbf{R} \mathbf{K}^{-1} \mathbf{p}_1 \\ \mathbf{p}_{2,min} &= \mathbf{p}_{2r} + \mathbf{Kt}\rho_{1,min} \\ \mathbf{p}_{2,max} &= \mathbf{p}_{2r} + \mathbf{Kt}\rho_{1,max}\end{aligned}$$

3.1. Non-Keyframes → Inverse Depth Update

ImmaturePoint.cpp::traceOn()::L97

```

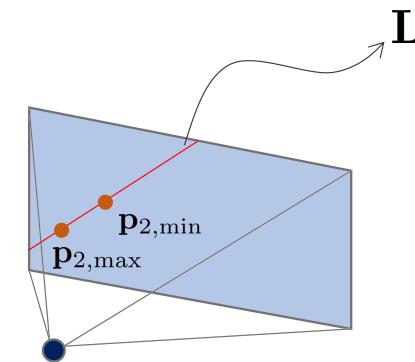
Vec3f pr = hostToFrame_KRKi * Vec3f(u,v, 1);
Vec3f ptpMin = pr + hostToFrame_Kt*idepth_min;
float uMin = ptpMin[0] / ptpMin[2];
float vMin = ptpMin[1] / ptpMin[2];
{...}
float dist;
float uMax;
float vMax;
Vec3f ptpMax;

if(std::isfinite(idepth_max))
{
    ptpMax = pr + hostToFrame_Kt*idepth_max;
    uMax = ptpMax[0] / ptpMax[2];
    vMax = ptpMax[1] / ptpMax[2];
    {...}
}
else
{
    dist = maxPixSearch;
    // project to arbitrary depth to get direction.
    ptpMax = pr + hostToFrame_Kt*0.01;
    uMax = ptpMax[0] / ptpMax[2];
    vMax = ptpMax[1] / ptpMax[2];
    // direction.
    float dx = uMax-uMin;
    float dy = vMax-vMin;
    float d = 1.0f / sqrtf(dx*dx+dy*dy);
    // set to [setting_maxPixSearch].
    uMax = uMin + dist*dx*d;
    vMax = vMin + dist*dy*d;
}

```

Epipolar line 방향 계산 및 maximum discrete search 범위 설정

$$\mathbf{L} := \{l_0 + \lambda[l_x \ l_y]^T\}$$



3.1. Non-Keyframes → Inverse Depth Update

ImmaturePoint.cpp::traceOn()::L186

```
// ===== compute error-bounds on result in pixel. if the new interval is not at
least 1/2 of the old, SKIP =====
float dx = setting_trace_stepsize*(uMax-uMin);
float dy = setting_trace_stepsize*(vMax-vMin);
float a = (Vec2f(dx,dy).transpose() * gradH * Vec2f(dx,dy));
float b = (Vec2f(dy,-dx).transpose() * gradH * Vec2f(dy,-dx));
float errorInPixel = 0.2f + 0.2f * (a+b) / a;

if(errorInPixel*setting_trace_minImprovementFactor > dist && std::isfinite(idepth_max))
{
    if(debugPrint)
        printf("NO SIGNIFICANT IMPROVEMENT (%f)!\n", errorInPixel);
    lastTraceUV = Vec2f(uMax+uMin, vMax+vMin)*0.5;
    lastTracePixelInterval=dist;
    return lastTraceStatus = ImmaturePointStatus::IPS_BADCONDITION;
}

if(errorInPixel >10) errorInPixel=10;
```

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/ImmaturePoint.cpp#L186>

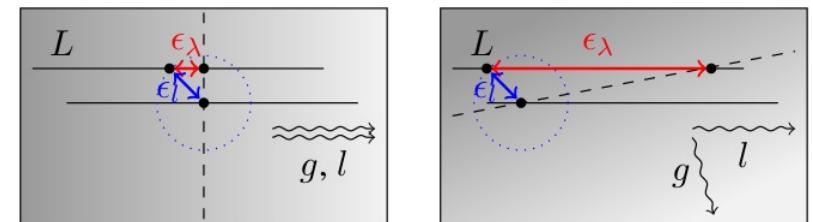
3.1. Non-Keyframes → Inverse Depth Update

ImmaturePoint.cpp::traceOn()::L186

```
// ===== compute error-bounds on result in pixel. if the new interval is not at
least 1/2 of the old, SKIP =====
float dx = setting_trace_stepsize*(uMax-uMin);
float dy = setting_trace_stepsize*(vMax-vMin);
float a = (Vec2f(dx,dy).transpose() * gradH * Vec2f(dx,dy));
float b = (Vec2f(dy,-dx).transpose() * gradH * Vec2f(dy,-dx));
float errorInPixel = 0.2f + 0.2f * (a+b) / a; Gradient 방향과 Epipolar Line의 방향을 측정

if(errorInPixel*setting_trace_minImprovementFactor > dist && std::isfinite(idepth_max))
{
    if(debugPrint)
        printf("NO SIGNIFICANT IMPROVEMENT (%f)!\n", errorInPixel);
    lastTraceUV = Vec2f(uMax+uMin, vMax+vMin)*0.5;
    lastTracePixelInterval=dist;
    return lastTraceStatus = ImmaturePointStatus::IPS_BADCONDITION;
}
```

$$\text{errorInPixel} = \alpha$$



<https://jsturm.de/publications/data/engel2013iccv.pdf> 논문 참조

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/ImmaturePoint.cpp#L186>

3.1. Non-Keyframes → Inverse Depth Update

ImmaturePoint.cpp::traceOn()::L186

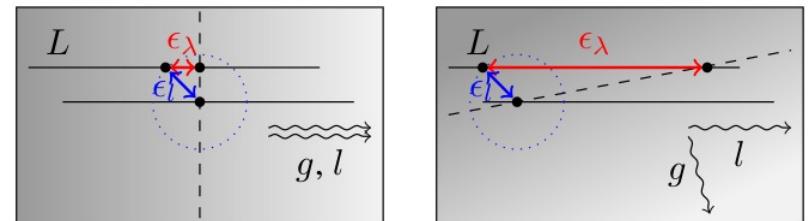
```
// ===== compute error-bounds on result in pixel. if the new interval is not at
least 1/2 of the old, SKIP =====
float dx = setting_trace_stepsize*(uMax-uMin);
float dy = setting_trace_stepsize*(vMax-vMin);
float a = (Vec2f(dx,dy).transpose() * gradH * Vec2f(dx,dy));
float b = (Vec2f(dy,-dx).transpose() * gradH * Vec2f(dy,-dx));
float errorInPixel = 0.2f + 0.2f * (a+b) / a;

if(errorInPixel*setting_trace_minImprovementFactor > dist && std::isfinite(idepth_max))
{
    if(debugPrint)
        printf("NO SIGNIFICANT IMPROVEMENT (%f)\n", errorInPixel);
    lastTraceUV = Vec2f(uMax+uMin, vMax+vMin)*0.5;
    lastTracePixelInterval=dist;
    return lastTraceStatus = ImmaturePointStatus::IPS_BADCONDITION;
}
```

해당 값이 1보다 많이 큰 경우 (오른쪽 그림에서 l,g가 수직인 경우)

에러가 매우 클 것으로 판단해서 더 이상 진행하지 않고 함수 종료

$$\text{errorInPixel} = \alpha$$



<https://jsturm.de/publications/data/engel2013iccv.pdf> 논문 참조

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/ImmaturePoint.cpp#L186>

3.1. Non-Keyframes → Inverse Depth Update

ImmaturePoint.cpp::traceOn()::L259

```

for(int i=0;i<numSteps;i++)
{
    float energy=0;
    for(int idx=0;idx<patternNum;idx++)
    {
        float hitColor = getInterpolatedElement31(frame->dI,
            (float)(ptx+rotatePattern[idx][0]),
            (float)(pty+rotatePattern[idx][1]),
            wG[0]);
        if(!std::isfinite(hitColor)) {energy+=1e5; continue;}
        float residual = hitColor - (float)(hostToFrame_affine[0] * color[idx] +
            hostToFrame_affine[1]);
        float hw = fabs(residual) < setting_huberTH ? 1 : setting_huberTH / fabs(residual);
        energy += hw *residual*residual*(2-hw);
    }
    if(debugPrint)
        printf("step %.1f %.1f (id %f): energy = %f!\n",
            ptx, pty, 0.0f, energy);

    errors[i] = energy;

    if(energy < bestEnergy)
    {
        bestU = ptx; bestV = pty; bestEnergy = energy; bestIdx = i;
    }
    ptx+=dx;
    pty+=dy;
}
// find best score outside a +-2px radius.
float secondBest=1e10;
for(int i=0;i<numSteps;i++)
{
    if((i < bestIdx-setting_minTraceTestRadius || i > bestIdx+setting_minTraceTestRadius) &&
        errors[i] < secondBest)
        secondBest = errors[i];
}
float newQuality = secondBest / bestEnergy;
if(newQuality < quality || numSteps > 10) quality = newQuality;

```

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/ImmaturePoint.cpp#L259>

3.1. Non-Keyframes → Inverse Depth Update

ImmaturePoint.cpp::traceOn()::L259

```

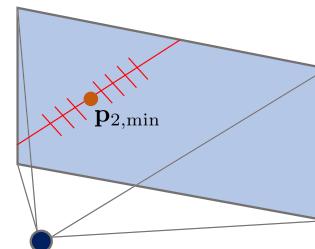
for(int i=0;i<numSteps;i++)
{
    float energy=0;
    for(int idx=0;idx<patternNum;idx++)
    {
        float hitColor = getInterpolatedElement31(frame->dI,
            (float)(ptx+rotatePattern[idx][0]),
            (float)(pty+rotatePattern[idx][1]),
            wG[0]);
        if(!std::isfinite(hitColor)) {energy+=1e5; continue;}
        float residual = hitColor - (float)(hostToFrame_affine[0] * color[idx] +
            hostToFrame_affine[1]);
        float hw = fabs(residual) < setting_hubertH ? 1 : setting_hubertH / fabs(residual);
        energy += hw *residual*residual*(2-hw);
    }
    if(debugPrint)
        printf("step %.1f %.1f (id %f): energy = %f!\n",
            ptx, pty, 0.0f, energy);

    errors[i] = energy;

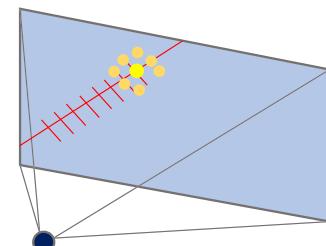
    if(energy < bestEnergy)
    {
        bestU = ptx; bestV = pty; bestEnergy = energy; bestIdx = i;
    }
    ptx+=dx;
    pty+=dy;
}
// find best score outside a +-2px radius.
float secondBest=1e10;
for(int i=0;i<numSteps;i++)
{
    if((i < bestIdx-setting_minTraceTestRadius || i > bestIdx+setting_minTraceTestRadius) &&
        errors[i] < secondBest)
        secondBest = errors[i];
}
float newQuality = secondBest / bestEnergy;
if(newQuality < quality || numSteps > 10) quality = newQuality;

```

discrete search를 수행하면서 밝기 오차를 errors[i] 배열에 저장



Set the discrete search range



Find two smallest errors

3.1. Non-Keyframes → Inverse Depth Update

ImmaturePoint.cpp::traceOn()::L259

```

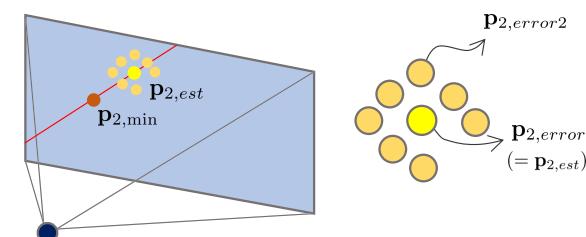
for(int i=0;i<numSteps;i++)
{
    float energy=0;
    for(int idx=0;idx<patternNum;idx++)
    {
        float hitColor = getInterpolatedElement31(frame->dI,
            (float)(ptx+rotatePattern[idx][0]),
            (float)(pty+rotatePattern[idx][1]),
            wG[0]);
        if(!std::isfinite(hitColor)) {energy+=1e5; continue;}
        float residual = hitColor - (float)(hostToFrame_affine[0] * color[idx] +
            hostToFrame_affine[1]);
        float hw = fabs(residual) < setting_huberTH ? 1 : setting_huberTH / fabs(residual);
        energy += hw * residual*residual*(2-hw);
    }
    if(debugPrint)
        printf("step %.1f %.1f (id %f): energy = %f!\n",
            ptx, pty, 0.0f, energy);

    errors[i] = energy;

    if(energy < bestEnergy)
    {
        bestU = ptx; bestV = pty; bestEnergy = energy; bestIdx = i;
    }
    ptx+=dx;
    pty+=dy;
}
// find best score outside a +-2px radius.
float secondBest=1e10;
for(int i=0;i<numSteps;i++)
{
    if((i < bestIdx-setting_minTraceTestRadius || i > bestIdx+setting_minTraceTestRadius) &&
    errors[i] < secondBest)
        secondBest = errors[i];
}
float newQuality = secondBest / bestEnergy;
if(newQuality < quality || numSteps > 10) quality = newQuality;

```

가장 작은 오차 2개를 선정한 후 두 오차의 비율을 비교하여 퀄리티 평가



3.1. Non-Keyframes → Inverse Depth Update

ImmaturePoint.cpp::traceOn()::L302

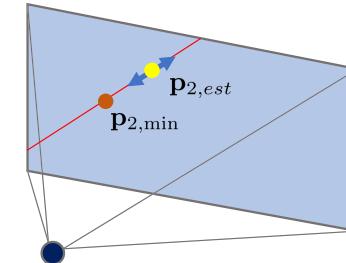
```
// ===== do GN optimization =====
float uBak=bestU, vBak=bestV, gnstepsize=1, stepBack=0;
if(setting_trace_GNIterations>0) bestEnergy = 1e5;
int gnStepsGood=0, gnStepsBad=0;
for(int it=0;it<setting_trace_GNIterations;it++) {
    float H = 1, b=0, energy=0;
    for(int idx=0;idx<patternNum;idx++) {
        Vec3f hitColor = getInterpolatedElement33(frame->dI,
        (float)(bestU+rotatePattern[idx][0]),
        (float)(bestV+rotatePattern[idx][1]),wG[0]);
        if(!std::isfinite((float)hitColor[0])) {energy+=1e5; continue;}
        float residual = hitColor[0] - (hostToFrame_affine[0] * color[idx] + hostToFrame_affine[1]);
        float dResdDist = dx*hitColor[1] + dy*hitColor[2];
        float hw = fabs(residual) < setting_huberTH ? 1 : setting_huberTH / fabs(residual);
        H += hw*dResdDist*dResdDist;
        b += hw*residual*dResdDist;
        energy += weights[idx]*weights[idx]*hw *residual*residual*(2-hw);
    }
}
```

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/ImmaturePoint.cpp#L302>

3.1. Non-Keyframes → Inverse Depth Update

ImmaturePoint.cpp::traceOn()::L302

```
// ===== do GN optimization =====
float uBak=bestU, vBak=bestV, gnstepsize=1, stepBack=0;
if(setting_trace_GNIterations>0) bestEnergy = 1e5;           GN optimization을 통해 최적의 위치 설정
int gnStepsGood=0, gnStepsBad=0;
for(int it=0;it<setting_trace_GNIterations;it++) {
    float H = 1, b=0, energy=0;
    for(int idx=0;idx<patternNum;idx++) {
        Vec3f hitColor = getInterpolatedElement33(frame->dI,
        (float)(bestU+rotatePattern[idx][0]),wG[0]),
        (float)(bestV+rotatePattern[idx][1]),wG[0]);
        if(!std::isfinite((float)hitColor[0])) {energy+=1e5; continue;}
        float residual = hitColor[0] - (hostToFrame_affine[0] * color[idx] + hostToFrame_affine[1]);
        float dResdDist = dx*hitColor[1] + dy*hitColor[2];
        float hw = fabs(residual) < setting_hubertH ? 1 : setting_hubertH / fabs(residual);
        H += hw*dResdDist*dResdDist;
        b += hw*residual*dResdDist;
        energy += weights[idx]*weights[idx]*hw *residual*residual*(2-hw);
    }
}
```



Perform Gauss-Newton optimization for refinement

3.1. Non-Keyframes → Inverse Depth Update

ImmaturePoint.cpp::traceOn()::L388

```
// ===== set new interval =====
if(dx*dx>dy*dy)
{
idepth_min = (pr[2]*(bestU-errorInPixel*dx) - pr[0]) / (hostToFrame_Kt[0] - hostToFrame_Kt[2]*(bestU-errorInPixel*dx));
idepth_max = (pr[2]*(bestU+errorInPixel*dx) - pr[0]) / (hostToFrame_Kt[0] - hostToFrame_Kt[2]*(bestU+errorInPixel*dx));
}
else
{
idepth_min = (pr[2]*(bestV-errorInPixel*dy) - pr[1]) / (hostToFrame_Kt[1] - hostToFrame_Kt[2]*(bestV-errorInPixel*dy));
idepth_max = (pr[2]*(bestV+errorInPixel*dy) - pr[1]) / (hostToFrame_Kt[1] - hostToFrame_Kt[2]*(bestV+errorInPixel*dy));
}
```

3.1. Non-Keyframes → Inverse Depth Update

ImmaturePoint.cpp::traceOn()::L388

```
// ===== set new interval =====
if(dx*dx>dy*dy)
{
idepth_min = (pr[2]*(bestU-errorInPixel*dx) - pr[0]) / (hostToFrame_Kt[0] - hostToFrame_Kt[2]*(bestU-errorInPixel*dx));
idepth_max = (pr[2]*(bestU+errorInPixel*dx) - pr[0]) / (hostToFrame_Kt[0] - hostToFrame_Kt[2]*(bestU+errorInPixel*dx));
}
else
{
idepth_min = (pr[2]*(bestV-errorInPixel*dy) - pr[1]) / (hostToFrame_Kt[1] - hostToFrame_Kt[2]*(bestV-errorInPixel*dy));
idepth_max = (pr[2]*(bestV+errorInPixel*dy) - pr[1]) / (hostToFrame_Kt[1] - hostToFrame_Kt[2]*(bestV+errorInPixel*dy));
}
```

Immature point의 새로운 idepth interval 계산 (=idepth 값 업데이트)

x gradient가 큰 경우 $\Delta u > \Delta v$

$$\rho_{1,\min} = \frac{m_3(u_2^* - \alpha\Delta u) - m_1}{n_1 - n_3(u_2^* - \alpha\Delta u)}$$

$$\rho_{1,\max} = \frac{m_3(u_2^* + \alpha\Delta u) - m_1}{n_1 - n_3(u_2^* + \alpha\Delta u)}$$

y gradient가 큰 경우 $\Delta u < \Delta v$

$$\rho_{1,\min} = \frac{m_3(v_2^* - \alpha\Delta v) - m_2}{n_2 - n_3(v_2^* - \alpha\Delta v)}$$

$$\rho_{1,\max} = \frac{m_3(v_2^* + \alpha\Delta v) - m_2}{n_2 - n_3(v_2^* + \alpha\Delta v)}$$

4.3.1. Keyframes → Sliding Window Optimization

RawResidualJacobian.h::L32

```

struct RawResidualJacobian
{
    EIGEN_MAKE_ALIGNED_OPERATOR_NEW;
    // ===== new structure: save independently =====.
    VecNRf resF;
    // the two rows of d[x,y]/d[xi].
    Vec6f Jpdxi[2]; // 2x6
    // the two rows of d[x,y]/d[C].
    VecCf JpdC[2]; // 2x4
    // the two rows of d[x,y]/d[idepth].
    Vec2f Jpdd; // 2x1
    // the two columns of d[r]/d[x,y].
    VecNRF JIdx[2]; // 9x2
    // = the two columns of d[r] / d[ab]
    VecNRF JabF[2]; // 9x2
    // = JIdx^T * JIdx (inner product). Only as a shorthand.
    Mat22f JIdx2; // 2x2
    // = Jab^T * JIdx (inner product). Only as a shorthand.
    Mat22f JabJIdx; // 2x2
    // = Jab^T * Jab (inner product). Only as a shorthand.
    Mat22f Jab2; // 2x2
};

```

각 변수들의 의미는 다음과 같다

$$\text{resF} \quad \mathbf{r}_{21} \in \mathbb{R}^{8 \times 1}$$

$$\text{Jpdxi}[2] \quad \frac{\partial \mathbf{p}_2}{\partial \xi_{21}} \in \mathbb{R}^{2 \times 6}$$

$$\text{JpdC}[2] \quad \frac{\partial \mathbf{p}_2}{\partial \mathbf{c}} \in \mathbb{R}^{2 \times 4}$$

$$\text{Jpdd} \quad \frac{\partial \mathbf{p}_2}{\partial \rho_1} \in \mathbb{R}^{2 \times 1}$$

$$\text{JIdx}[2] \quad \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{p}_2} \in \mathbb{R}^{8 \times 2}$$

$$\text{JabF}[2] \quad \frac{\partial \mathbf{r}_{21}}{\partial a_{21}}, \frac{\partial \mathbf{r}_{21}}{\partial b_{21}} \in \mathbb{R}^{8 \times 1} \text{ each}$$

$$\text{JIdx2} \quad \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{p}_2}^T \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{p}_2} \in \mathbb{R}^{2 \times 2}$$

$$\text{JabJIdx} \quad \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_{21}}^T \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{p}_2} \in \mathbb{R}^{2 \times 2}$$

$$\text{Jab2} \quad \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_{21}}^T \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_{21}} \in \mathbb{R}^{2 \times 2}$$

$$\begin{aligned} \mathbf{c} &= [f_x \quad f_y \quad c_x \quad c_y]^T \\ \mathbf{l}_{21} &= [a_{21} \quad b_{21}]^T \end{aligned}$$

<https://github.com/JakobEngel/dso/blob/master/src/OptimizationBackend/RawResidualJacobian.h#L32>

4.3.1. Keyframes → Sliding Window Optimization

CODE REVIEW

Residuals.cpp::linearize()

각 변수들의 의미는 다음과 같다

```
float drescale, u, v, new_idepth;
float Ku, Kv;
Vec3f KliP;

if(!projectPoint(point->u, point->v, point->idepth_zero_scaled, 0, 0, HCalib,
PRE_RTll_0, PRE_tTll_0, drescale, u, v, Ku, Kv, KliP, new_idepth))
{ state_NewState = ResState::OOB; return state_energy; }

d_d_x = drescale * (PRE_tTll_0[0]-PRE_tTll_0[2]*u)*SCALE_IDEPTH*HCalib->fxl();
d_d_y = drescale * (PRE_tTll_0[1]-PRE_tTll_0[2]*v)*SCALE_IDEPTH*HCalib->fyi();
```

```
d_C_x[2] = drescale*(PRE_RTll_0(2,0)*u-PRE_RTll_0(0,0));
d_C_x[3] = HCalib->fxl() * drescale*(PRE_RTll_0(2,1)*u-PRE_RTll_0(0,1)) * HCalib->fyli();
d_C_x[0] = KliP[0]*d_C_x[2];
d_C_x[1] = KliP[1]*d_C_x[3];

d_C_y[2] = HCalib->fyi() * drescale*(PRE_RTll_0(2,0)*v-PRE_RTll_0(1,0)) * HCalib->fxli();
d_C_y[3] = drescale*(PRE_RTll_0(2,1)*v-PRE_RTll_0(1,1));
d_C_y[0] = KliP[0]*d_C_y[2];
d_C_y[1] = KliP[1]*d_C_y[3];

d_C_x[0] = (d_C_x[0]+u)*SCALE_F;
d_C_x[1] *= SCALE_F;
d_C_x[2] = (d_C_x[2]+1)*SCALE_C;
d_C_x[3] *= SCALE_C;

d_C_y[0] *= SCALE_F;
d_C_y[1] = (d_C_y[1]+v)*SCALE_F;
d_C_y[2] *= SCALE_C;
d_C_y[3] = (d_C_y[3]+1)*SCALE_C;
```

$$\text{KliP} \quad \mathbf{K}^{-1}\mathbf{p}_1 = \bar{\mathbf{p}}_1$$

$$\text{ptp} \quad \mathbf{R}_{21}\mathbf{K}^{-1}\mathbf{p}_1 + \rho_1\mathbf{t}_{21} = \rho_2^{-1}\rho_1\mathbf{K}^{-1}\mathbf{p}_2$$

$$\text{drescale} \quad \rho_2\rho_1^{-1}$$

$$[\mathbf{u}, \mathbf{v}, 1]^T$$

$$\bar{\mathbf{p}}_2 = \mathbf{K}^{-1}\mathbf{p}_2$$

$$\mathbf{p}_2 = \mathbf{K}\bar{\mathbf{p}}_2 = \mathbf{K}\rho_2(\mathbf{R}_{21}\rho_1^{-1}\mathbf{K}^{-1}\mathbf{p}_1 + \mathbf{t}_{21})$$

$$\frac{\partial \mathbf{p}_2}{\partial \rho_1} \in \mathbb{R}^{2 \times 1} \text{ 를 구하는 코드}$$

Jpdd

$$\frac{\partial \mathbf{p}_2}{\partial \mathbf{c}} \in \mathbb{R}^{2 \times 4} \text{ 를 구하는 코드}$$

(유도식은 4.3.2 섹션 참조)

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/Residuals.cpp#L78>

4.3.1. Keyframes → Sliding Window Optimization

Residuals.cpp::linearize() 각 변수들의 의미는 다음과 같다

```
d_xi_x[0] = new_idepth*HCalib->fxl();
d_xi_x[1] = 0;
d_xi_x[2] = -new_idepth*u*HCalib->fxl();
d_xi_x[3] = -u*v*HCalib->fxl();
d_xi_x[4] = (1+u*u)*HCalib->fxl();
d_xi_x[5] = -v*v*HCalib->fxl();
```

```
d_xi_y[0] = 0;
d_xi_y[1] = new_idepth*HCalib->fyl();
d_xi_y[2] = -new_idepth*v*HCalib->fyl();
d_xi_y[3] = -(1+v*v)*HCalib->fyl();
d_xi_y[4] = u*v*HCalib->fyl();
d_xi_y[5] = u*u*HCalib->fyl();
```

Jpdxi[2] $\frac{\partial \mathbf{p}_2}{\partial \xi_{21}} \in \mathbb{R}^{2 \times 6}$ 를 구하는 코드

(유도식은 1.3 섹션 참조)

```
J->JIdx[0][idx] = hitColor[1];
J->JIdx[1][idx] = hitColor[2];
```

JIdx[2] $\frac{\partial \mathbf{r}_{21}}{\partial \mathbf{p}_2} \in \mathbb{R}^{8 \times 2}$ 를 구하는 코드

$$\frac{\partial \mathbf{r}_{21}}{\partial \mathbf{p}_2} = w_h \frac{\partial \mathbf{I}_2(\mathbf{p}_2)}{\partial \mathbf{p}_2} = w_h [\nabla \mathbf{I}_x \quad \nabla \mathbf{I}_y]$$

```
float drdA = (color[idx]-b0);
J->JabF[0][idx] = drdA*hw;
J->JabF[1][idx] = hw;
```

JabF[2] $\frac{\partial \mathbf{r}_{21}}{\partial a_{21}}, \frac{\partial \mathbf{r}_{21}}{\partial b_{21}} \in \mathbb{R}^{8 \times 1}$ each 를 구하는 코드

$$\begin{aligned}\frac{\partial \mathbf{r}_{21}}{\partial a_{21}} &= -w_h \exp(a) \mathbf{I}_1(b_0 - \mathbf{p}_1) \\ \frac{\partial \mathbf{r}_{21}}{\partial b_{21}} &= -w_h\end{aligned}$$

4.3.1. Keyframes → Sliding Window Optimization

CODE REVIEW

EnergyFunctionalStructs.cpp::takeDataF()

각 변수들의 의미는 다음과 같다

```
void EFResidual::takeDataF()
{
    std::swap<RawResidualJacobian*>(J, data->J);
    Vec2f JI_JI_Jd = J->JIIdx2 * J->Jpdd;

    for(int i=0;i<6;i++)
        JpJdF[i] = J->Jpdx[0][i]*JI_JI_Jd[0] + J->Jpdx[1][i] * JI_JI_Jd[1];

    JpJdF.segment<2>(6) = J->JabJIIdx*J->Jpdd;
}
```

$$Jl_Jl_Jd \quad \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{p}_2}^T \frac{\partial \mathbf{r}_{21}}{\partial \rho_1} \in \mathbb{R}^{2 \times 1}$$

를 구하는 코드 $(2 \times 8)(8 \times 1) = (2 \times 1)$

$$JpJdF.segment<6>(0) \quad \frac{\partial \mathbf{p}_2}{\partial \xi_{21}}^T \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{p}_2}^T \frac{\partial \mathbf{r}_{21}}{\partial \rho_1} = \frac{\partial \mathbf{r}_{21}}{\partial \xi_{21}}^T \frac{\partial \mathbf{r}_{21}}{\partial \rho_1} \in \mathbb{R}^{6 \times 1}$$

$(6 \times 2)(2 \times 8)(8 \times 1) = (6 \times 1)$

$$JpJdF.segment<2>(6) \quad \frac{\partial \mathbf{r}_{21}}{\partial l_{21}}^T \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{p}_2} \frac{\partial \mathbf{p}_2}{\partial \rho_1} = \frac{\partial \mathbf{r}_{21}}{\partial l_{21}}^T \frac{\partial \mathbf{r}_{21}}{\partial \rho_1} \in \mathbb{R}^{2 \times 1}$$

$(2 \times 8)(8 \times 2)(2 \times 1) = (2 \times 1)$

$$JpJdF \quad \begin{bmatrix} \frac{\partial \mathbf{r}_{21}}{\partial \xi_{21}}^T \frac{\partial \mathbf{r}_{21}}{\partial \rho_1} \\ \frac{\partial \mathbf{r}_{21}}{\partial l_{21}}^T \frac{\partial \mathbf{r}_{21}}{\partial \rho_1} \end{bmatrix} \in \mathbb{R}^{8 \times 1}$$

<https://github.com/JakobEngel/dso/blob/master/src/OptimizationBackend/EnergyFunctionalStructs.cpp#L39>

4.3.1. Keyframes → Sliding Window Optimization

CODE REVIEW

AccumulatedTopHessian.cpp::addPoint()

각 변수들의 의미는 다음과 같다

```

VecNRF resApprox;

if(mode==0) // active
    resApprox = rJ->resF;
if(mode==2) // marginalize
    resApprox = r->res_toZeroF;
if(mode==1) // linearized
{
    // compute Jp*delta
    __m128 Jp_delta_x = _mm_set1_ps(rJ->Jpdx[0]).dot(dp.head<6>())+rJ->Jpdc[0].dot(dc)+rJ->Jpdd[0]*dd;
    __m128 Jp_delta_y = _mm_set1_ps(rJ->Jpdx[1]).dot(dp.head<6>())+rJ->Jpdc[1].dot(dc)+rJ->Jpdd[1]*dd;
    __m128 delta_a = _mm_set1_ps((float)(dp[6]));
    __m128 delta_b = _mm_set1_ps((float)(dp[7]));

    for(int i=0;i<patternNum;i+=4)
    {
        // PATTERN: rtz = resF - [JI*Jp Ja]*delta.
        __m128 rtz = _mm_load_ps(((float*)&r->res_toZeroF)+i);
        rtz = _mm_add_ps(rtz,_mm_mul_ps(_mm_load_ps(((float*)(rJ->JIdx))+i),Jp_delta_x));
        rtz = _mm_add_ps(rtz,_mm_mul_ps(_mm_load_ps(((float*)(rJ->JIdx+1))+i),Jp_delta_y));
        rtz = _mm_add_ps(rtz,_mm_mul_ps(_mm_load_ps(((float*)(rJ->JabF))+i),delta_a));
        rtz = _mm_add_ps(rtz,_mm_mul_ps(_mm_load_ps(((float*)(rJ->JabF+1))+i),delta_b));
        _mm_store_ps(((float*)&resApprox)+i, rtz);
    }
}

```

$$[\begin{aligned} & Jp_delta_x \\ & Jp_delta_y \end{aligned}] = \frac{\partial \mathbf{p}_2}{\partial \xi_1} \delta \xi_1 + \frac{\partial \mathbf{p}_2}{\partial \xi_2} \delta \xi_2 + \frac{\partial \mathbf{p}_2}{\partial \mathbf{c}} \delta \mathbf{c} + \frac{\partial \mathbf{p}_2}{\partial \mathbf{p}_1} \delta \mathbf{p}_1$$

$$rtz = \frac{\partial \mathbf{p}_2}{\partial \xi_1} \delta \xi_1 + \frac{\partial \mathbf{p}_2}{\partial \xi_2} \delta \xi_2 + \frac{\partial \mathbf{p}_2}{\partial \mathbf{c}} \delta \mathbf{c} + \frac{\partial \mathbf{p}_2}{\partial \mathbf{p}_1} \delta \mathbf{p}_1 + \frac{\partial \mathbf{p}_2}{\partial \mathbf{l}_1} \delta \mathbf{l}_1 + \frac{\partial \mathbf{p}_2}{\partial \mathbf{l}_2} \delta \mathbf{l}_2$$

$$[\begin{aligned} & \text{delta_a} \\ & \text{delta_b} \end{aligned}] = \frac{\partial \mathbf{p}_2}{\partial \mathbf{l}_1} \delta \mathbf{l}_1 + \frac{\partial \mathbf{p}_2}{\partial \mathbf{l}_2} \delta \mathbf{l}_2$$

$$\text{resApprox}(\text{mode}0) \quad \mathbf{r}_{21} \in \mathbb{R}^{8 \times 1}$$

$$\text{resApprox}(\text{mode}2) \quad \mathbf{r}_{21} + rtz$$

resApprox(mode1)은 코드 구현이 되어 있으나 실제로 해당 코드를 사용하지 않는다.

(즉, mode 값이 1인 경우가 발생하지 않는다.)

$$\mathbf{r}_{21} + rtz$$

4.3.1. Keyframes → Sliding Window Optimization

AccumulatedTopHessian.cpp::addPoint()

각 변수들의 의미는 다음과 같다

```
// need to compute JI^T * r, and Jab^T * r. (both are 2-vectors).
Vec2f JI_r[0,0];
Vec2f Jab_r[0,0];
float rr=0;

for(int i=0;i<patternNum;i++)
{
    JI_r[0] += resApprox[i] * rJ->JIIdx[0][i];
    JI_r[1] += resApprox[i] * rJ->JIIdx[1][i];
    Jab_r[0] += resApprox[i] * rJ->JabF[0][i];
    Jab_r[1] += resApprox[i] * rJ->JabF[1][i];
    rr += resApprox[i]*resApprox[i];
}
```

```
acc[tid][htIDX].update(
    rJ->JpdC[0].data(), rJ->Jpdxi[0].data(),
    rJ->JpdC[1].data(), rJ->Jpdxi[1].data(),
    rJ->JIIdx2(0,0), rJ->JIIdx2(0,1), rJ->JIIdx2(1,1));
acc[tid][htIDX].updateBotRight(
    rJ->Jab2(0,0), rJ->Jab2(0,1), Jab_r[0],
    rJ->Jab2(1,1), Jab_r[1], rr);
acc[tid][htIDX].updateTopRight(
    rJ->JpdC[0].data(), rJ->Jpdxi[0].data(),
    rJ->JpdC[1].data(), rJ->Jpdxi[1].data(),
    rJ->JabJIdx(0,0), rJ->JabJIdx(0,1),
    rJ->JabJIdx(1,0), rJ->JabJIdx(1,1),
    JI_r[0], JI_r[1]);
```

Jl_r

$$\sum \left(\frac{\partial \mathbf{r}_{21}}{\partial \mathbf{p}_2} \right)^T \mathbf{r}_{21}$$

Jab_r

$$\sum \left(\frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_{21}} \right)^T \mathbf{r}_{21}$$

rr

$$\mathbf{r}_{21}^T \mathbf{r}_{21}$$

acc

$$\mathbf{H}_{yy}^A = \begin{bmatrix} \mathbf{J}^T \\ \mathbf{r}^T \end{bmatrix} [\mathbf{J} \quad \mathbf{r}] = \begin{bmatrix} \mathbf{H}_a & \mathbf{H}_b \\ \mathbf{H}_b^T & \mathbf{H}_c \end{bmatrix}_{13 \times 13}$$

acc.update()

$$\mathbf{H}_a = \begin{bmatrix} \left(\frac{\partial \mathbf{r}_{21}}{\partial \mathbf{c}} \right)^T \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{c}} & 4 \times 4 & \left(\frac{\partial \mathbf{r}_{21}}{\partial \mathbf{c}} \right)^T \frac{\partial \mathbf{r}_{21}}{\partial \xi_{21}} & 4 \times 6 \\ \left(\frac{\partial \mathbf{r}_{21}}{\partial \xi_{21}} \right)^T \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{c}} & 6 \times 4 & \left(\frac{\partial \mathbf{r}_{21}}{\partial \xi_{21}} \right)^T \frac{\partial \mathbf{r}_{21}}{\partial \xi_{21}} & 6 \times 6 \end{bmatrix}_{10 \times 10}$$

acc.updateBotRight()

$$\mathbf{H}_c = \begin{bmatrix} \left(\frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_{21}} \right)^T \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_{21}} & 2 \times 2 & \left(\frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_{21}} \right)^T \mathbf{r}_{21,2 \times 1} \\ \mathbf{r}_{21}^T \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_{21}} & 1 \times 2 & \mathbf{r}_{21}^T \mathbf{r}_{21,1 \times 1} \end{bmatrix}_{3 \times 3}$$

acc.updateTopRight()

$$\mathbf{H}_b = \begin{bmatrix} \left(\frac{\partial \mathbf{r}_{21}}{\partial \mathbf{c}} \right)^T \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_{21}} & 4 \times 2 & \left(\frac{\partial \mathbf{r}_{21}}{\partial \mathbf{c}} \right)^T \mathbf{r}_{21,4 \times 1} \\ \left(\frac{\partial \mathbf{r}_{21}}{\partial \xi_{21}} \right)^T \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_{21}} & 6 \times 2 & \left(\frac{\partial \mathbf{r}_{21}}{\partial \xi_{21}} \right)^T \mathbf{r}_{21,6 \times 1} \end{bmatrix}_{10 \times 3}$$

$$\mathbf{J} = \begin{bmatrix} \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{c}} & \frac{\partial \mathbf{r}_{21}}{\partial \xi_{21}} & \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_{21}} \end{bmatrix}_{8 \times 12}$$

$$\mathbf{r} = [\mathbf{r}_{21}]_{8 \times 1}$$

4.3.1. Keyframes → Sliding Window Optimization

CODE REVIEW

AccumulatedTopHessian.cpp::addPoint()

각 변수들의 의미는 다음과 같다

```
Vec2f Ji2_Jpdd = rJ->JI_idx2 * rJ->Jpdd;
bd_acc += JI_r[0]*rJ->Jpdd[0] + JI_r[1]*rJ->Jpdd[1];
Hdd_acc += Ji2_Jpdd.dot(rJ->Jpdd);
Hcd_acc += rJ->Jpdc[0]*Ji2_Jpdd[0] + rJ->Jpdc[1]*Ji2_Jpdd[1];
```

Ji2_Jpdd	$\frac{\partial \mathbf{r}_{21}}{\partial \mathbf{p}_2}^T \frac{\partial \mathbf{r}_{21}}{\partial \rho_1} \in \mathbb{R}^{2 \times 1}$
bd_acc	$\frac{\partial \mathbf{r}_{21}}{\partial \rho_1}^T \mathbf{r}_{21} \in \mathbb{R}^{1 \times 1}$ active
	marginalize
	$\frac{\partial \mathbf{r}_{21}}{\partial \rho_1}^T (\frac{\partial \mathbf{p}_2}{\partial \xi_1} \delta \xi_1 + \frac{\partial \mathbf{p}_2}{\partial \xi_2} \delta \xi_2 + \frac{\partial \mathbf{p}_2}{\partial \mathbf{c}} \delta \mathbf{c} + \frac{\partial \mathbf{p}_2}{\partial \mathbf{p}_1} \delta \mathbf{p}_1 + \frac{\partial \mathbf{p}_2}{\partial \mathbf{l}_1} \delta \mathbf{l}_1 + \frac{\partial \mathbf{p}_2}{\partial \mathbf{l}_2} \delta \mathbf{l}_2)$
Hdd_acc	$\frac{\partial \mathbf{r}_{21}}{\partial \rho_1}^T \frac{\partial \mathbf{r}_{21}}{\partial \rho_1} \in \mathbb{R}^{1 \times 1}$
Hcd_acc	$\frac{\partial \mathbf{r}_{21}}{\partial \mathbf{c}}^T \frac{\partial \mathbf{r}_{21}}{\partial \rho_1} \in \mathbb{R}^{4 \times 1}$

<https://github.com/JakobEngel/dso/blob/master/src/OptimizationBackend/AccumulatedTopHessian.cpp#L40>

4.3.1. Keyframes → Sliding Window Optimization

CODE REVIEW

AccumulatedTopHessian.cpp::stitchDoubleInternal()

모든 키프레임에서 host, target에 대한 H,b 행렬을 구하는 함수

$$\mathbf{H}_{\mathbf{y}\mathbf{y}} = \mathbf{J}^T \mathbf{J} = \begin{bmatrix} \mathbf{J}_{\mathbf{c}}^T \mathbf{J}_{\mathbf{c}} & \sum \mathbf{J}_{\mathbf{c}}^T \mathbf{J}_{\xi'} \\ \sum \mathbf{J}_{\xi'}^T \mathbf{J}_{\mathbf{c}} & \sum \mathbf{J}_{\xi'}^T \mathbf{J}_{\xi'} \end{bmatrix}_{68 \times 68} =$$

$$\begin{aligned}\mathbf{y} &= [\delta\xi^T \ a \ b \ \mathbf{c}] \\ \xi' &= [\delta\xi \ a \ b]_{8 \times 1}\end{aligned}$$

A 9x9 grid heatmap illustrating the relationship between various components. The rows and columns are labeled as follows:

- Rows: K(4), KF#1(8), KF#2(8), KF#3(8), KF#4(8), KF#5(8), KF#6(8), KF#7(8), KF#8(8)
- Columns: K(4), KF#1(8), KF#2(8), KF#3(8), KF#4(8), KF#5(8), KF#6(8), KF#7(8), KF#8(8)

The heatmap uses a color scale where the top-left cell (K(4), K(4)) is red, and all other cells are dark blue. This indicates that K(4) is unique or has no significant overlap with the other components listed.

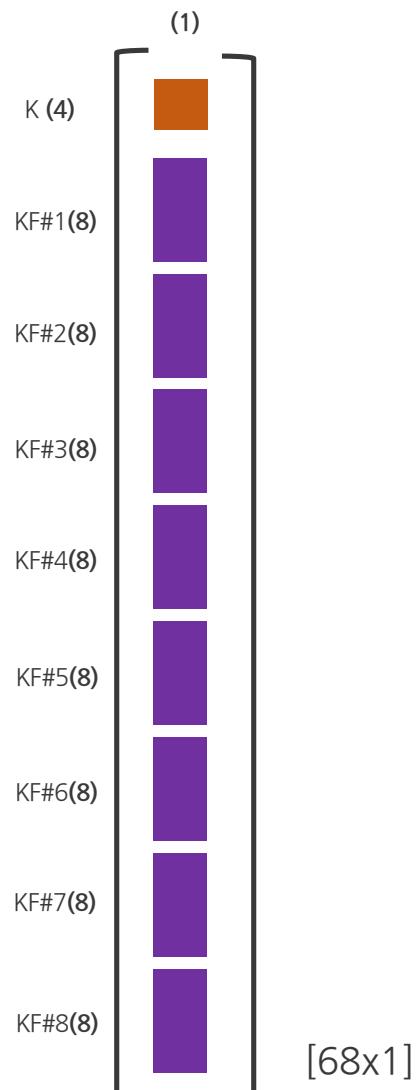
4.3.1. Keyframes → Sliding Window Optimization

CODE REVIEW

AccumulatedTopHessian.cpp::stitchDoubleInternal()

모든 키프레임에서 host, target에 대한 H,b 행렬을 구하는 함수

$$\mathbf{b}_y = -\mathbf{J}^T \mathbf{r}_{21} = \begin{bmatrix} -\mathbf{J}_c^T \mathbf{r}_{21} \\ -\sum \mathbf{J}_{\xi'}^T \mathbf{r}_{21} \end{bmatrix}_{68 \times 1} =$$



$$\mathbf{y} = [\delta\xi^T \ a \ b \ \mathbf{c}]$$
$$\xi' = [\delta\xi \ a \ b]_{8 \times 1}$$

4.3.1. Keyframes → Sliding Window Optimization

AccumulatedTopHessian.cpp::stitchDoubleInternal()

각 변수들의 의미는 다음과 같다

```

for(int k=min;k<max;k++)
{
    int h = k%nframes[0];
    int t = k/nframes[0];
    int hIdx = CPARS+h*8;
    int tIdx = CPARS+t*8;
    int aidx = h+nframes[0]*t;

    assert(aidx == k);
    MatPCPC accH = MatPCPC::Zero();

    for(int tid2=0;tid2 < toAggregate;tid2++)
    {
        acc[tid2][aidx].finish();
        if(acc[tid2][aidx].num==0)
            continue;
        accH += acc[tid2][aidx].H.cast<double>();
    }
}

```

$$\begin{bmatrix} \left(\frac{\partial \mathbf{r}_{21}}{\partial \xi_1}\right)^T \frac{\partial \mathbf{r}_{21}}{\partial \xi_1} & \left(\frac{\partial \mathbf{r}_{21}}{\partial \xi_1}\right)^T \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_1} \\ \left(\frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_1}\right)^T \frac{\partial \mathbf{r}_{21}}{\partial \xi_1} & \left(\frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_1}\right)^T \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_1} \end{bmatrix} = \begin{bmatrix} \left(\frac{\partial \xi_{21}}{\partial \xi_1}\right)^T & \\ & \left(\frac{\partial \mathbf{l}_{21}}{\partial \mathbf{l}_1}\right)^T \end{bmatrix} \begin{bmatrix} \left(\frac{\partial \mathbf{r}_{21}}{\partial \xi_{21}}\right)^T \frac{\partial \mathbf{r}_{21}}{\partial \xi_{21}} & \left(\frac{\partial \mathbf{r}_{21}}{\partial \xi_{21}}\right)^T \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_{21}} \\ \left(\frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_{21}}\right)^T \frac{\partial \mathbf{r}_{21}}{\partial \xi_{21}} & \left(\frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_{21}}\right)^T \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_{21}} \end{bmatrix} \begin{bmatrix} \frac{\partial \xi_{21}}{\partial \xi_1} & \\ & \frac{\partial \mathbf{l}_{21}}{\partial \mathbf{l}_1} \end{bmatrix}$$

$$\begin{bmatrix} \left(\frac{\partial \mathbf{r}_{21}}{\partial \xi_2}\right)^T \frac{\partial \mathbf{r}_{21}}{\partial \xi_2} & \left(\frac{\partial \mathbf{r}_{21}}{\partial \xi_2}\right)^T \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_1} \\ \left(\frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_1}\right)^T \frac{\partial \mathbf{r}_{21}}{\partial \xi_2} & \left(\frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_1}\right)^T \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_1} \end{bmatrix} = \begin{bmatrix} \left(\frac{\partial \xi_{21}}{\partial \xi_2}\right)^T & \\ & \left(\frac{\partial \mathbf{l}_{21}}{\partial \mathbf{l}_1}\right)^T \end{bmatrix} \begin{bmatrix} \left(\frac{\partial \mathbf{r}_{21}}{\partial \xi_{21}}\right)^T \frac{\partial \mathbf{r}_{21}}{\partial \xi_{21}} & \left(\frac{\partial \mathbf{r}_{21}}{\partial \xi_{21}}\right)^T \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_{21}} \\ \left(\frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_{21}}\right)^T \frac{\partial \mathbf{r}_{21}}{\partial \xi_{21}} & \left(\frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_{21}}\right)^T \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_{21}} \end{bmatrix} \begin{bmatrix} \frac{\partial \xi_{21}}{\partial \xi_2} & \\ & \frac{\partial \mathbf{l}_{21}}{\partial \mathbf{l}_1} \end{bmatrix}$$

$$\begin{bmatrix} \left(\frac{\partial \mathbf{r}_{21}}{\partial \xi_1}\right)^T \frac{\partial \mathbf{r}_{21}}{\partial \xi_2} & \left(\frac{\partial \mathbf{r}_{21}}{\partial \xi_1}\right)^T \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_2} \\ \left(\frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_1}\right)^T \frac{\partial \mathbf{r}_{21}}{\partial \xi_2} & \left(\frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_1}\right)^T \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_2} \end{bmatrix} = \begin{bmatrix} \left(\frac{\partial \xi_{21}}{\partial \xi_1}\right)^T & \\ & \left(\frac{\partial \mathbf{l}_{21}}{\partial \mathbf{l}_1}\right)^T \end{bmatrix} \begin{bmatrix} \left(\frac{\partial \mathbf{r}_{21}}{\partial \xi_{21}}\right)^T \frac{\partial \mathbf{r}_{21}}{\partial \xi_{21}} & \left(\frac{\partial \mathbf{r}_{21}}{\partial \xi_{21}}\right)^T \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_{21}} \\ \left(\frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_{21}}\right)^T \frac{\partial \mathbf{r}_{21}}{\partial \xi_{21}} & \left(\frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_{21}}\right)^T \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_{21}} \end{bmatrix} \begin{bmatrix} \frac{\partial \xi_{21}}{\partial \xi_2} & \\ & \frac{\partial \mathbf{l}_{21}}{\partial \mathbf{l}_2} \end{bmatrix}$$

```

H[tid].block<8,8>(hIdx, hIdx).noalias() += EF->adHost[aidx] * accH.block<8,8>(CPARS,CPARS) * EF->adHost[aidx].transpose();
H[tid].block<8,8>(tIdx, tIdx).noalias() += EF->adTarget[aidx] * accH.block<8,8>(CPARS,CPARS) * EF->adTarget[aidx].transpose();
H[tid].block<8,8>(hIdx, tIdx).noalias() += EF->adHost[aidx] * accH.block<8,8>(CPARS,CPARS) * EF->adTarget[aidx].transpose();
H[tid].block<8,CPARS>(hIdx,0).noalias() += EF->adHost[aidx] * accH.block<8,CPARS>(CPARS,0);
H[tid].block<8,CPARS>(tIdx,0).noalias() += EF->adTarget[aidx] * accH.block<8,CPARS>(CPARS,0);
H[tid].topLeftCorner<CPARS,CPARS>().noalias() += accH.block<CPARS,CPARS>(0,0);

b[tid].segment<8>(hIdx).noalias() += EF->adHost[aidx] * accH.block<8,1>(CPARS,CPARS+8);
b[tid].segment<8>(tIdx).noalias() += EF->adTarget[aidx] * accH.block<8,1>(CPARS,CPARS+8);
b[tid].head<CPARS>().noalias() += accH.block<CPARS,1>(0,CPARS+8);
}

```

1: host, 2: target

<https://github.com/JakobEngel/dso/blob/master/src/OptimizationBackend/AccumulatedTopHessian.cpp#L241>

4.3.1. Keyframes → Sliding Window Optimization

AccumulatedTopHessian.cpp::stitchDoubleInternal()

각 변수들의 의미는 다음과 같다

```

for(int k=min;k<max;k++)
{
    int h = k%nframes[0];
    int t = k/nframes[0];
    int hIdx = CPARS+h*8;
    int tIdx = CPARS+t*8;
    int aidx = h+nframes[0]*t;

    assert(aidx == k);
    MatPCPC accH = MatPCPC::Zero();

    for(int tid2=0;tid2 < toAggregate;tid2++)
    {
        acc[tid2][aidx].finish();
        if(acc[tid2][aidx].num==0)
            continue;
        accH += acc[tid2][aidx].H.cast<double>();

        H[tid].block<8,8>(hIdx, hIdx).noalias() += EF->adHost[aidx] * accH.block<8,8>(CPARS,CPARS) * EF->adHost[aidx].transpose();
        H[tid].block<8,8>(tIdx, tIdx).noalias() += EF->adTarget[aidx] * accH.block<8,8>(CPARS,CPARS) * EF->adTarget[aidx].transpose();
        H[tid].block<8,8>(hIdx, tIdx).noalias() += EF->adHost[aidx] * accH.block<8,8>(CPARS,CPARS) * EF->adTarget[aidx].transpose();
        H[tid].block<8,CPARS>(hIdx,0).noalias() += EF->adHost[aidx] * accH.block<8,CPARS>(CPARS,0);
        H[tid].block<8,CPARS>(tIdx,0).noalias() += EF->adTarget[aidx] * accH.block<8,CPARS>(CPARS,0);
        H[tid].topLeftCorner<CPARS,CPARS>().noalias() += accH.block<CPARS,CPARS>(0,0);

        b[tid].segment<8>(hIdx).noalias() += EF->adHost[aidx] * accH.block<8,1>(CPARS,CPARS+8);
        b[tid].segment<8>(tIdx).noalias() += EF->adTarget[aidx] * accH.block<8,1>(CPARS,CPARS+8);
        b[tid].head<CPARS>().noalias() += accH.block<CPARS,1>(0,CPARS+8);
    }
}

```

$$\begin{bmatrix} \left(\frac{\partial \mathbf{r}_{21}}{\partial \xi_1}\right)^T \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{c}} \\ \left(\frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_1}\right)^T \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{c}} \end{bmatrix} = \begin{bmatrix} \left(\frac{\partial \xi_{21}}{\partial \xi_1}\right)^T \\ \left(\frac{\partial \mathbf{l}_{21}}{\partial \mathbf{l}_1}\right)^T \end{bmatrix} \begin{bmatrix} \left(\frac{\partial \mathbf{r}_{21}}{\partial \xi_{21}}\right)^T \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{c}} \\ \left(\frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_{21}}\right)^T \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{c}} \end{bmatrix}$$

$$\begin{bmatrix} \left(\frac{\partial \mathbf{r}_{21}}{\partial \xi_2}\right)^T \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{c}} \\ \left(\frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_1}\right)^T \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{c}} \end{bmatrix} = \begin{bmatrix} \left(\frac{\partial \xi_{21}}{\partial \xi_2}\right)^T \\ \left(\frac{\partial \mathbf{l}_{21}}{\partial \mathbf{l}_1}\right)^T \end{bmatrix} \begin{bmatrix} \left(\frac{\partial \mathbf{r}_{21}}{\partial \xi_{21}}\right)^T \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{c}} \\ \left(\frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_{21}}\right)^T \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{c}} \end{bmatrix}$$

$$H[tid].topLeftCorner \quad \left[\left(\frac{\partial \mathbf{r}_{21}}{\partial \mathbf{c}} \right)^T \frac{\partial \mathbf{r}_{21}}{\partial \mathbf{c}} \right] \in \mathbb{R}^{4 \times 4}$$

1: host, 2: target

<https://github.com/JakobEngel/dso/blob/master/src/OptimizationBackend/AccumulatedTopHessian.cpp#L241>

4.3.1. Keyframes → Sliding Window Optimization

AccumulatedTopHessian.cpp::stitchDoubleInternal()

각 변수들의 의미는 다음과 같다

```

for(int k=min;k<max;k++)
{
    int h = k%nframes[0];
    int t = k/nframes[0];
    int hIdx = CPARS+h*8;
    int tIdx = CPARS+t*8;
    int aidx = h+nframes[0]*t;

    assert(aidx == k);
    MatPCPC accH = MatPCPC::Zero();

    for(int tid2=0;tid2 < toAggregate;tid2++)
    {
        acc[tid2][aidx].finish();
        if(acc[tid2][aidx].num==0)
            continue;
        accH += acc[tid2][aidx].H.cast<double>();

        H[tid].block<8,8>(hIdx, hIdx).noalias() += EF->adHost[aidx] * accH.block<8,8>(CPARS,CPARS) * EF->adHost[aidx].transpose();
        H[tid].block<8,8>(tIdx, tIdx).noalias() += EF->adTarget[aidx] * accH.block<8,8>(CPARS,CPARS) * EF->adTarget[aidx].transpose();
        H[tid].block<8,8>(hIdx, tIdx).noalias() += EF->adHost[aidx] * accH.block<8,8>(CPARS,CPARS) * EF->adTarget[aidx].transpose();
        H[tid].block<8,CPARS>(hIdx,0).noalias() += EF->adHost[aidx] * accH.block<8,CPARS>(CPARS,0);
        H[tid].block<8,CPARS>(tIdx,0).noalias() += EF->adTarget[aidx] * accH.block<8,CPARS>(CPARS,0);
        H[tid].topLeftCorner<CPARS,CPARS>().noalias() += accH.block<CPARS,CPARS>(0,0);

        b[tid].segment<8>(hIdx).noalias() += EF->adHost[aidx] * accH.block<8,1>(CPARS,CPARS+8);
        b[tid].segment<8>(tIdx).noalias() += EF->adTarget[aidx] * accH.block<8,1>(CPARS,CPARS+8);
        b[tid].head<CPARS>().noalias() += accH.block<CPARS,1>(0,CPARS+8);
    }
}

```

$$\begin{bmatrix} \left(\frac{\partial \mathbf{r}_{21}}{\partial \xi_1}\right)^T \mathbf{r}_{21} \\ \left(\frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_1}\right)^T \mathbf{r}_{21} \end{bmatrix} = \begin{bmatrix} \left(\frac{\partial \xi_{21}}{\partial \xi_1}\right)^T \\ \left(\frac{\partial \mathbf{l}_{21}}{\partial \mathbf{l}_1}\right)^T \end{bmatrix} \begin{bmatrix} \left(\frac{\partial \mathbf{r}_{21}}{\partial \xi_{21}}\right)^T \mathbf{r}_{21} \\ \left(\frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_{21}}\right)^T \mathbf{r}_{21} \end{bmatrix}$$

$$\begin{bmatrix} \left(\frac{\partial \mathbf{r}_{21}}{\partial \xi_2}\right)^T \mathbf{r}_{21} \\ \left(\frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_1}\right)^T \mathbf{r}_{21} \end{bmatrix} = \begin{bmatrix} \left(\frac{\partial \xi_{21}}{\partial \xi_2}\right)^T \\ \left(\frac{\partial \mathbf{l}_{21}}{\partial \mathbf{l}_1}\right)^T \end{bmatrix} \begin{bmatrix} \left(\frac{\partial \mathbf{r}_{21}}{\partial \xi_{21}}\right)^T \mathbf{r}_{21} \\ \left(\frac{\partial \mathbf{r}_{21}}{\partial \mathbf{l}_{21}}\right)^T \mathbf{r}_{21} \end{bmatrix}$$

b[tid].head $\left[\left(\frac{\partial \mathbf{r}_{21}}{\partial \mathbf{c}}\right)^T \mathbf{r}_{21}\right] \in \mathbb{R}^{4 \times 1}$

1: host, 2: target

<https://github.com/JakobEngel/dso/blob/master/src/OptimizationBackend/AccumulatedTopHessian.cpp#L241>

4.3.1. Keyframes → Sliding Window Optimization

CODE REVIEW

AccumulatedSCHessian.cpp::addPoint()

각 변수들의 의미는 다음과 같다

```
float H = p->Hdd_accAF+p->Hdd_accLF+p->priorF;
if(H < 1e-10) H = 1e-10;

p->data->idepth_hessian=H;
p->HdiF = 1.0 / H;
p->bdSumF = p->bd_accAF + p->bd_accLF;

if(shiftPriorToZero) p->bdSumF += p->priorF*p->deltaF;

VecCf Hcd = p->Hcd_accAF + p->Hcd_accLF;
```

H

$$\sum_{i=1}^N \left(\frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}}^T \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right) \in \mathbb{R}^{1 \times 1}$$

p->HdiF

$$\sum_{i=1}^N \left(\frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}}^T \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right)^{-1} \in \mathbb{R}^{1 \times 1}$$

p->bdSumF

$$\sum_{i=1}^N \left(\frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}}^T \mathbf{r}_{21}^{(i)} \right) \in \mathbb{R}^{1 \times 1}$$

Hcd

$$\sum_{i=1}^N \left(\frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \mathbf{c}}^T \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right) \in \mathbb{R}^{4 \times 1}$$

1: host, 2: target

<https://github.com/JakobEngel/dso/blob/master/src/OptimizationBackend/AccumulatedSCHessian.cpp#L34>

4.3.1. Keyframes → Sliding Window Optimization

CODE REVIEW

AccumulatedSCHessian.cpp::stitchDoubleInternal()

모든 키프레임에서 host, target에 대한
Schur Complement H,b 행렬을 구하는 함수

$$\mathbf{H}_{sc} = \mathbf{H}_{\mathbf{y}\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{H}_{\rho\mathbf{y}} = \sum \frac{1}{\mathbf{J}_\rho^T \mathbf{J}_\rho} (\mathbf{J}_\rho^T \mathbf{J}_\mathbf{y})^T \mathbf{J}_\rho^T \mathbf{J}_\mathbf{y} = \quad \text{KF#4(8)}$$

$$\mathbf{y} = [\delta\xi^T \ a \ b \ \mathbf{c}]$$

$$\mathbf{H}_{\rho\rho} = \sum \mathbf{J}_\rho^T \mathbf{J}_\rho$$

$$\mathbf{H}_{\rho\mathbf{y}} = \sum \mathbf{J}_\rho^T \mathbf{J}_{\mathbf{y}}$$

$$\mathbf{H}_{\mathbf{y}\rho} = \sum \mathbf{J}_{\mathbf{y}}^T \mathbf{J}_{\rho}$$

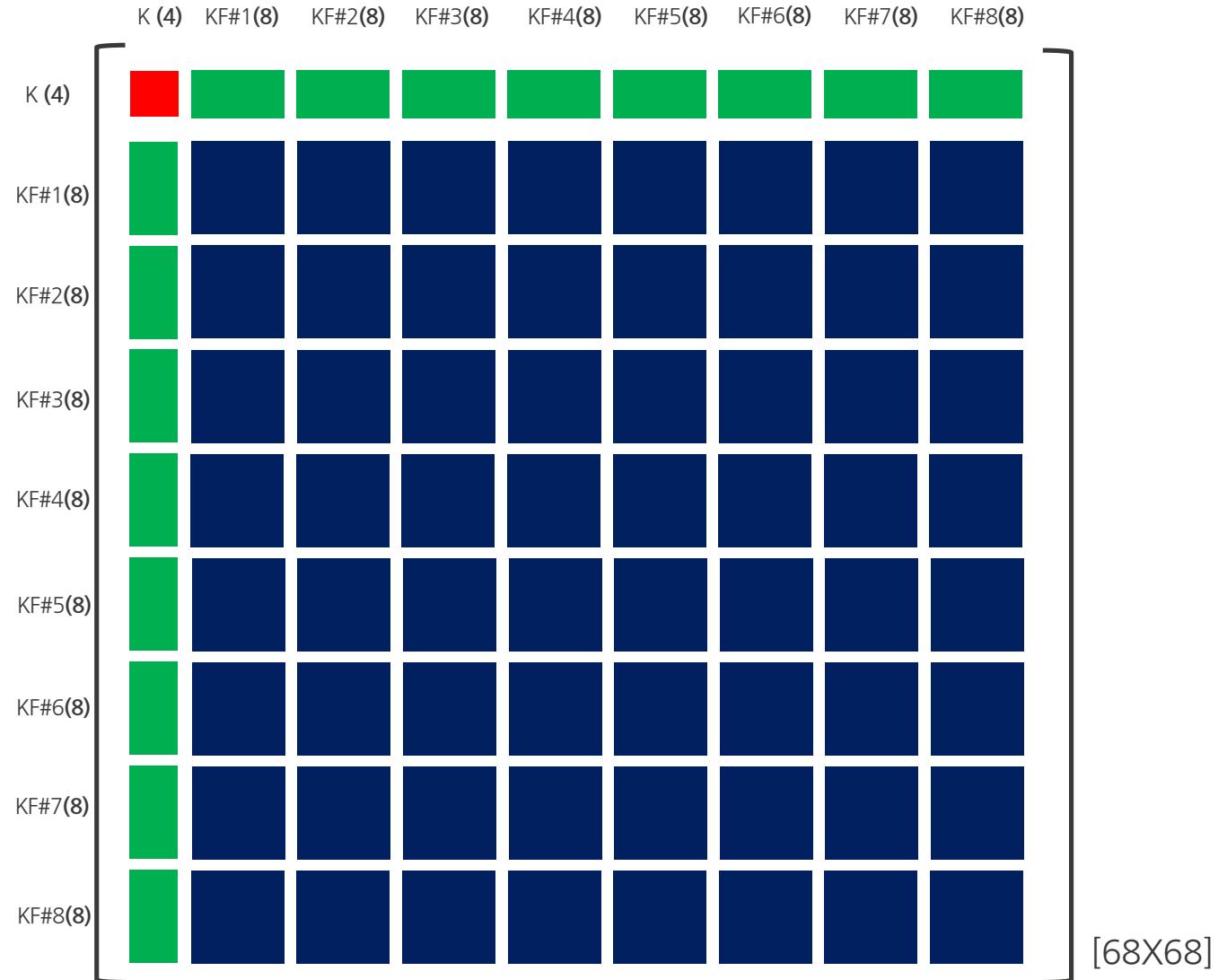
$$\mathbf{H}_{yy} = \sum \mathbf{J}_y^T \mathbf{J}_y$$

$$\mathbf{b}_\rho = - \sum \mathbf{J}_\rho^T \mathbf{r}_{21}$$

$$\mathbf{b}_y = - \sum \mathbf{J}_y^T \mathbf{r}_{21}$$

$$\mathbf{H}_\phi = \mathbf{H}_{\mathbf{v}\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{H}_{\rho\mathbf{v}}$$

$$\mathbf{b}_\phi = \mathbf{H}_{\mathbf{y}\rho} \mathbf{H}_{\theta\theta}^{-1} \mathbf{b}_\rho$$



4.3.1. Keyframes → Sliding Window Optimization

CODE REVIEW

AccumulatedSCHessian.cpp:stitchDoubleInternal()

모든 키프레임에서 host, target에 대한
Schur Complement H,b 행렬을 구하는 함수

$$\mathbf{b}_{sc} = \mathbf{H}_{\mathbf{y}\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{b}_\rho = - \sum \frac{1}{\mathbf{J}_\rho \mathbf{J}_\rho^T} (\mathbf{J}_\rho^T \mathbf{J}_\mathbf{y})^T \mathbf{J}_\rho^T \mathbf{r}_{21} = \text{KF#4(8)}$$

$$\mathbf{y} = [\delta\xi^T \ a \ b \ \mathbf{c}]$$

$$\mathbf{H}_{\rho\rho} = \sum \mathbf{J}_\rho^T \mathbf{J}_\rho$$

$$\mathbf{H}_{\rho\mathbf{y}} = \sum \mathbf{J}_\rho^T \mathbf{J}_\mathbf{y}$$

$$\mathbf{H}_{\mathbf{y}\rho} = \sum \mathbf{J}_\mathbf{y}^T \mathbf{J}_\rho$$

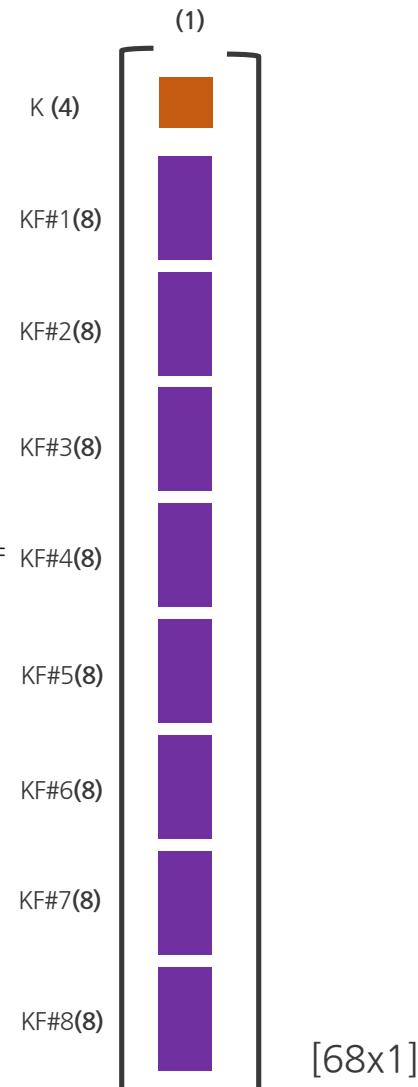
$$\mathbf{H}_{\mathbf{y}\mathbf{y}} = \sum \mathbf{J}_\mathbf{y}^T \mathbf{J}_\mathbf{y}$$

$$\mathbf{b}_\rho = - \sum \mathbf{J}_\rho^T \mathbf{r}_{21}$$

$$\mathbf{b}_\mathbf{y} = - \sum \mathbf{J}_\mathbf{y}^T \mathbf{r}_{21}$$

$$\mathbf{H}_\phi = \mathbf{H}_{\mathbf{y}\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{H}_{\rho\mathbf{y}}$$

$$\mathbf{b}_\phi = \mathbf{H}_{\mathbf{y}\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{b}_\rho$$



4.3.1. Keyframes → Sliding Window Optimization

AccumulatedSCHessian.cpp::addPoint()

각 변수들의 의미는 다음과 같다

```

accHcc[tid].update(Hcd,Hcd,p->HdiF);
accbc[tid].update(Hcd, p->bdSumF * p->HdiF);

assert(std::isfinite((float)(p->HdiF)));

int nFrames2 = nframes[tid]*nframes[tid];

for(EFResidual* r1 : p->residualsAll)
{
    if(!r1->isActive()) continue;
    int r1ht = r1->hostIDX + r1->targetIDX*nframes[tid];
    for(EFResidual* r2 : p->residualsAll)
    {
        if(!r2->isActive()) continue;
        accD[tid][r1ht+r2->targetIDX*nFrames2].update(r1->JpJdF, r2->JpJdF, p->HdiF);
    }
    accE[tid][r1ht].update(r1->JpJdF, Hcd, p->HdiF);
    accEB[tid][r1ht].update(r1->JpJdF,p->HdiF*p->bdSumF);
}
    
```

accHcc

$$\frac{1}{\mathbf{J}_\rho \mathbf{J}_\rho^T} (\mathbf{J}_\rho^T \mathbf{J}_c)^T \mathbf{J}_\rho^T \mathbf{J}_c = \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \mathbf{c}}^T \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right) \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}}^T \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right)^{-1} \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \mathbf{c}}^T \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right)^T$$

accbc

$$\frac{1}{\mathbf{J}_\rho \mathbf{J}_\rho^T} (\mathbf{J}_\rho^T \mathbf{J}_c)^T \mathbf{J}_\rho^T \mathbf{r}_{21} = \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \mathbf{c}}^T \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right) \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}}^T \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right)^{-1} \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \mathbf{c}}^T \mathbf{r}_{21}^{(i)} \right)$$

accD

$$\frac{1}{\mathbf{J}_\rho \mathbf{J}_\rho^T} (\mathbf{J}_\rho^T \mathbf{J}_{\xi'})^T \mathbf{J}_\rho^T \mathbf{J}_{\xi'} = \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \xi'_{21}}^T \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right) \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}}^T \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right)^{-1} \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \xi'_{31}}^T \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right)^T$$

accE

$$\frac{1}{\mathbf{J}_\rho \mathbf{J}_\rho^T} (\mathbf{J}_\rho^T \mathbf{J}_{\xi'})^T \mathbf{J}_\rho^T \mathbf{J}_c = \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \xi'_{21}}^T \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right) \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}}^T \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right)^{-1} \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \mathbf{c}}^T \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right)^T$$

accEB

$$\frac{1}{\mathbf{J}_\rho \mathbf{J}_\rho^T} (\mathbf{J}_\rho^T \mathbf{J}_{\xi'})^T \mathbf{J}_\rho^T \mathbf{r}_{21} = \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \xi'_{21}}^T \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right) \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}}^T \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right)^{-1} \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}}^T \mathbf{r}_{21}^{(i)} \right)^T$$

1: host, 2: target(1), 3: target(2)

$$\left(\frac{\partial \xi'_{21}}{\partial \xi'_1} \right) = \begin{bmatrix} \left(\frac{\partial \xi_{21}}{\partial \xi_1} \right)^T & \\ & \left(\frac{\partial \mathbf{l}_{21}}{\partial \mathbf{l}_1} \right)^T \end{bmatrix}$$

$$\xi' = [\delta \xi \ a \ b]_{8 \times 1} \text{ pose}(6) + \text{photo}(2)$$

4.3.1. Keyframes → Sliding Window Optimization

AccumulatedSCHessian.cpp:stitchDoubleInternal()

```

Mat8C Hpc = Mat8C::Zero();
Vec8 bp = Vec8::Zero();
for(int tid2=0;tid2 < toAggregate;tid2++)
{
    accE[tid2][ijIdx].finish();
    accEB[tid2][ijIdx].finish();
    Hpc += accE[tid2][ijIdx].A1m.cast<double>();
    bp += accEB[tid2][ijIdx].A1m.cast<double>();
}

```

```

H[tid].block<8,CPARS>(iIdx,0) += EF->adHost[ijIdx] * Hpc;
H[tid].block<8,CPARS>(jIdx,0) += EF->adTarget[ijIdx] * Hpc;
b[tid].segment<8>(iIdx) += EF->adHost[ijIdx] * bp;
b[tid].segment<8>(jIdx) += EF->adTarget[ijIdx] * bp;

```

```

for(int k=0;k<nf;k++)
{
    int kIdx = CPARS+k*8;
    int ijkIdx = ijIdx + k*nframes2;
    int ikIdx = i+nf*k;
    Mat88 accDM = Mat88::Zero();
    for(int tid2=0;tid2 < toAggregate;tid2++)
    {
        accD[tid2][ijkIdx].finish();
        if(accD[tid2][ijkIdx].num == 0) continue;
        accDM += accD[tid2][ijkIdx].A1m.cast<double>();
    }
    H[tid].block<8,8>(iIdx, iIdx) += EF->adHost[ijIdx] * accDM * EF->adHost[ikIdx].transpose();
    H[tid].block<8,8>(jIdx, kIdx) += EF->adTarget[ijIdx] * accDM * EF->adTarget[ikIdx].transpose();
    H[tid].block<8,8>(jIdx, iIdx) += EF->adTarget[ijIdx] * accDM * EF->adHost[ikIdx].transpose();
    H[tid].block<8,8>(iIdx, kIdx) += EF->adHost[ijIdx] * accDM * EF->adTarget[ikIdx].transpose();
}

```

각 변수들의 의미는 다음과 같다

$$\sum_{j=1}^M \left(\frac{\partial \xi'_{21}}{\partial \xi'_1} \right) \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \xi'_{21}}^T \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right) \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}}^T \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right)^{-1} \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \mathbf{c}}^T \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right)^T$$

$$\sum_{j=1}^M \left(\frac{\partial \xi'_{21}}{\partial \xi'_2} \right) \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \xi'_{21}}^T \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right) \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}}^T \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right)^{-1} \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \mathbf{c}}^T \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right)^T$$

$$\sum_{j=1}^M \left(\frac{\partial \xi'_{21}}{\partial \xi'_1} \right) \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \xi'_{21}}^T \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right) \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}}^T \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right)^{-1} \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}}^T \mathbf{r}_{21}^{(i)} \right)^T$$

$$\sum_{j=1}^M \left(\frac{\partial \xi'_{21}}{\partial \xi'_2} \right) \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \xi'_{21}}^T \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right) \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}}^T \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right)^{-1} \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}}^T \mathbf{r}_{21}^{(i)} \right)^T$$

1: host, 2: target(1), 3: target(2)

$$\left(\frac{\partial \xi'_{21}}{\partial \xi'_1} \right) = \begin{bmatrix} \left(\frac{\partial \xi_{21}}{\partial \xi_1} \right)^T & \\ & \left(\frac{\partial \mathbf{l}_{21}}{\partial \mathbf{l}_1} \right)^T \end{bmatrix}$$

4.3.1. Keyframes → Sliding Window Optimization

AccumulatedSCHessian.cpp:stitchDoubleInternal()

각 변수들의 의미는 다음과 같다

```

Mat8C Hpc = Mat8C::Zero();
Vec8 bp = Vec8::Zero();
for(int tid2=0;tid2 < toAggregate;tid2++)
{
    accE[tid2][ijIdx].finish();
    accEB[tid2][ijIdx].finish();
    Hpc += accE[tid2][ijIdx].A1m.cast<double>();
    bp += accEB[tid2][ijIdx].A1m.cast<double>();
}

H[tid].block<8,CPARS>(iIdx,0) += EF->adHost[ijIdx] * Hpc;
H[tid].block<8,CPARS>(jIdx,0) += EF->adTarget[ijIdx] * Hpc;
b[tid].segment<8>(iIdx) += EF->adHost[ijIdx] * bp;
b[tid].segment<8>(jIdx) += EF->adTarget[ijIdx] * bp;

for(int k=0;k<nf;k++)
{
    int ijkIdx = CPARS+k*8;
    int ijkId = ijkIdx + k*nframes2;
    int ikId = i+nf*k;
    Mat88 accDM = Mat88::Zero();
    for(int tid2=0;tid2 < toAggregate;tid2++)
    {
        accD[tid2][ijkId].finish();
        if(accD[tid2][ijkId].num == 0) continue;
        accDM += accD[tid2][ijkId].A1m.cast<double>();
    }

    H[tid].block<8,8>(iIdx, iIdx) += EF->adHost[ijIdx] * accDM * EF->adHost[ikIdx].transpose();
    H[tid].block<8,8>(jIdx, kIdx) += EF->adTarget[ijIdx] * accDM * EF->adTarget[ikIdx].transpose();
    H[tid].block<8,8>(jIdx, iIdx) += EF->adTarget[ijIdx] * accDM * EF->adHost[ikIdx].transpose();
    H[tid].block<8,8>(iIdx, kIdx) += EF->adHost[ijIdx] * accDM * EF->adTarget[ikIdx].transpose();
}

```

EF->adHost[ijIdx]

$$\sum_{j=1}^M \left(\frac{\partial \xi'_{21}}{\partial \xi'_1} \right) \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \xi'_{21}} \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right) \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right)^{-1} \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \xi'_{31}} \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right)^T \left(\frac{\partial \xi'_{31}}{\partial \xi'_1} \right)^T$$

EF->adTarget[ijIdx]

$$\sum_{j=1}^M \left(\frac{\partial \xi'_{21}}{\partial \xi'_2} \right) \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \xi'_{21}} \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right) \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right)^{-1} \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \xi'_{31}} \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right)^T \left(\frac{\partial \xi'_{31}}{\partial \xi'_2} \right)^T$$

EF->adTarget[ijIdx]

$$\sum_{j=1}^M \left(\frac{\partial \xi'_{21}}{\partial \xi'_2} \right) \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \xi'_{21}} \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right) \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right)^{-1} \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \xi'_{31}} \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right)^T \left(\frac{\partial \xi'_{31}}{\partial \xi'_2} \right)^T$$

EF->adHost[ijIdx]

$$\sum_{j=1}^M \left(\frac{\partial \xi'_{21}}{\partial \xi'_1} \right) \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \xi'_{21}} \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right) \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right)^{-1} \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \xi'_{31}} \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right)^T \left(\frac{\partial \xi'_{31}}{\partial \xi'_1} \right)^T$$

EF->adTarget[ikIdx]

1: host, 2: target(1), 3: target(2)

$$\left(\frac{\partial \xi'_{21}}{\partial \xi'_1} \right) = \begin{bmatrix} \left(\frac{\partial \xi_{21}}{\partial \xi_1} \right)^T & \\ & \left(\frac{\partial \mathbf{l}_{21}}{\partial \mathbf{l}_1} \right)^T \end{bmatrix}$$

4.3.1. Keyframes → Sliding Window Optimization

CODE REVIEW

AccumulatedSCHessian.cpp:stitchDoubleInternal()

각 변수들의 의미는 다음과 같다 1: host, 2: target(1), 3: target(2)

```
if(min==0)
{
    for(int tid2=0;tid2 < toAggregate;tid2++)
    {
        accHcc[tid2].finish();
        accbc[tid2].finish();
        H[tid].topLeftCorner<CPARS,CPARS>() += accHcc[tid2].A1m.cast<double>();
        b[tid].head<CPARS>() += accbc[tid2].A1m.cast<double>();
    }
}
```

H.topLeftCorner

$$\sum_{j=1}^M \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \mathbf{c}}^T \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right) \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}}^T \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right)^{-1} \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \mathbf{c}}^T \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right)^T$$

b.head

$$\sum_{j=1}^M \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \mathbf{c}}^T \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right) \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}}^T \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho_1^{(j)}} \right)^{-1} \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \mathbf{c}}^T \mathbf{r}_{21}^{(i)} \right)$$

4.3.1. Keyframes → Sliding Window Optimization

EnergyFunctional.cpp::solveSystemF()

다음을 순서대로 구한다.

```

MatXX HL_top, HA_top, H_sc;
VecX bL_top, bA_top, bM_top, b_sc;

accumulateAF_MT(HA_top, bA_top,multiThreading);
accumulateLF_MT(HL_top, bL_top,multiThreading);
accumulateSCF_MT(H_sc, b_sc,multiThreading);

bM_top = (bM+ HM * getStitchedDeltaF());

MatXX HFinal_top;
VecX bFinal_top;

if(setting_solveMode & SOLVER_ORTHOGONALIZE_SYSTEM)
{
    // have a look if prior is there.
    bool haveFirstFrame = false;
    for(EFFrame* f : frames) if(f->frameID==0) haveFirstFrame=true;
    MatXX HT_act = HL_top + HA_top - H_sc;
    VecX bT_act = bL_top + bA_top - b_sc;
    if(!haveFirstFrame)
        orthogonalize(&bT_act, &HT_act);
    HFinal_top = HT_act + HM;
    bFinal_top = bT_act + bM_top;
    lastHS = HFinal_top;
    lastbS = bFinal_top;
    for(int i=0;i<8*nFrames+CPARS;i++) HFinal_top(i,i) *= (1+lambda);
}
else
{
    HFinal_top = HL_top + HM + HA_top;
    bFinal_top = bL_top + bM_top + bA_top - b_sc;
    lastHS = HFinal_top - H_sc;
    lastbS = bFinal_top;
    for(int i=0;i<8*nFrames+CPARS;i++) HFinal_top(i,i) *= (1+lambda);
    HFinal_top -= H_sc * (1.0f/(1+lambda));
}

```

H_{yy}^A, b_y^A

H_{yy}^L, b_y^L

H_{sc}, b_{sc}

b_y^M

A: active, L: linearized, M: marginalized, SC: Schur Complement

setting_solveMode = SOLVER_FIX_LAMBDA | SOLVER_ORTHOGONALIZE_X_LATER

<https://github.com/JakobEngel/dso/blob/master/src/OptimizationBackend/EnergyFunctional.cpp#L775>

4.3.1. Keyframes → Sliding Window Optimization

CODE REVIEW

EnergyFunctional.cpp::solveSystemF()

```
MatXX HL_top, HA_top, H_sc;
VecX bL_top, bA_top, bM_top, b_sc;

accumulateAF_MT(HA_top, bA_top,multiThreading);
accumulateLF_MT(HL_top, bL_top,multiThreading);
accumulateSCF_MT(H_sc, b_sc,multiThreading);

bM_top = (bM+ HM * getStitchedDeltaF());

MatXX HFinal_top;
VecX bFinal_top;

if(setting_solveMode & SOLVER_ORTHOGONALIZE_SYSTEM)
{
    // have a look if prior is there.
    bool haveFirstFrame = false;
    for(EFFrame* f : frames) if(f->frameID==0) haveFirstFrame=true;
    MatXX HT_act = HL_top + HA_top - H_sc;
    VecX bT_act = bL_top + bA_top - b_sc;
    if(!haveFirstFrame)
        orthogonalize(&bT_act, &HT_act);
    HFinal_top = HT_act + HM;
    bFinal_top = bT_act + bM_top;
    lastHS = HFinal_top;
    lastbS = bFinal_top;
    for(int i=0;i<8*nFrames+CPARS;i++) HFinal_top(i,i) *= (1+lambda);
}
else
{
    HFinal_top = HL_top + HM + HA_top;
    bFinal_top = bL_top + bM_top + bA_top - b_sc;
    lastHS = HFinal_top - H_sc;
    lastbS = bFinal_top;
    for(int i=0;i<8*nFrames+CPARS;i++) HFinal_top(i,i) *= (1+lambda);
    HFinal_top -= H_sc * (1.0f/(1+lambda));
}
```

setting_solveMode에 따라 최종 H,b를 다른 방식으로 구한다.

필자는

setting_solveMode = SOLVER_FIX_LAMBDA | SOLVER_ORTHOGONALIZE_X_LATER

인 기본 설정만 사용했으므로 해당하지 않는 설정 코드는 스킵한다.

A: active, L: linearized, M: marginalized, SC: Schur Complement

setting_solveMode = SOLVER_FIX_LAMBDA | SOLVER_ORTHOGONALIZE_X_LATER

<https://github.com/JakobEngel/dso/blob/master/src/OptimizationBackend/EnergyFunctional.cpp#L775>

4.3.1. Keyframes → Sliding Window Optimization

CODE REVIEW

EnergyFunctional.cpp::solveSystemF()

```

MatXX HL_top, HA_top, H_sc;
VecX bL_top, bA_top, bM_top, b_sc;

accumulateAF_MT(HA_top, bA_top,multiThreading);
accumulateLF_MT(HL_top, bL_top,multiThreading);
accumulateSCF_MT(H_sc, b_sc,multiThreading);

bM_top = (bM+ HM * getStitchedDeltaF());

MatXX HFinal_top;
VecX bFinal_top;

if(setting_solverMode & SOLVER_ORTHOGONALIZE_SYSTEM)
{
    // have a look if prior is there.
    bool haveFirstFrame = false;
    for(EFFrame* f : frames) if(f->frameID==0) haveFirstFrame=true;
    MatXX HT_act = HL_top + HA_top - H_sc;
    VecX bT_act = bL_top + bA_top - b_sc;
    if(!haveFirstFrame)
        orthogonalize(&bT_act, &HT_act);
    HFinal_top = HT_act + HM;
    bFinal_top = bT_act + bM_top;
    lastHS = HFinal_top;
    lastbS = bFinal_top;
    for(int i=0;i<8*nFrames+CPARS;i++) HFinal_top(i,i) *= (1+lambda);
}
else
{
    HFinal_top = HL_top + HM + HA_top;
    bFinal_top = bL_top + bM_top + bA_top - b_sc;
    lastHS = HFinal_top - H_sc;
    lastbS = bFinal_top;
    for(int i=0;i<8*nFrames+CPARS;i++) HFinal_top(i,i) *= (1+lambda);
    HFinal_top -= H_sc * (1.0f/(1+lambda));
}

```

다음을 순서대로 계산한다

$$H_{yy} = H_{yy}^A + H_{yy}^L + H_{yy}^M$$

$$b_z = b_y^A + b_y^L + b_y^M - b_{sc}$$

$$H_{zz} = (1 + \lambda)H_{yy} - \frac{1}{1+\lambda}H_{sc}$$

A: active, L: linearized, M: marginalized, SC: Schur Complement

setting_solveMode = SOLVER_FIX_LAMBDA | SOLVER_ORTHOGONALIZE_X_LATER

<https://github.com/JakobEngel/dso/blob/master/src/OptimizationBackend/EnergyFunctional.cpp#L775>

4.3.1. Keyframes → Sliding Window Optimization

CODE REVIEW

EnergyFunctional.cpp::solveSystemF()

```
VecX x; 업데이트 값 x 선언
if(setting_solveMode & SOLVER_SVD)
{
    VecX SVecI = HFinal_top.diagonal().cwiseSqrt().cwiseInverse();
    MatXX HFinalScaled = SVecI.asDiagonal() * HFinal_top * SVecI.asDiagonal();
    VecX bFinalScaled = SVecI.asDiagonal() * bFinal_top;
    Eigen::JacobiSVD<MatXX> svd(HFinalScaled, Eigen::ComputeThinU | Eigen::ComputeThinV);
    VecX S = svd.singularValues();

    double minSv = 1e10, maxSv = 0;

    for(int i=0;i<S.size();i++)
    {
        if(S[i] < minSv) minSv = S[i];
        if(S[i] > maxSv) maxSv = S[i];
    }
    VecX Ub = svd.matrixU().transpose()*bFinalScaled;
    int setZero=0;
    for(int i=0;i<Ub.size();i++)
    {
        if(S[i] < setting_solverModeDelta*maxSv)
        { Ub[i] = 0; setZero++; }

        if((setting_solveMode & SOLVER_SVD_CUT) && (i >= Ub.size()-7))
        { Ub[i] = 0; setZero++; }
        else Ub[i] /= S[i];
    }
    x = SVecI.asDiagonal() * svd.matrixV() * Ub;
}

else
{
    VecX SVecI = (HFinal_top.diagonal() + VecX::Constant(HFinal_top.cols(),
    10)).cwiseSqrt().cwiseInverse();
    MatXX HFinalScaled = SVecI.asDiagonal() * HFinal_top * SVecI.asDiagonal();
    x = SVecI.asDiagonal() * HFinalScaled.ldlt().solve(SVecI.asDiagonal() * bFinal_top);
}
```

setting_solveMode에 따라 최종 H,b를 다른 방식으로 구한다.

필자는

setting_solveMode = SOLVER_FIX_LAMBDA | SOLVER_ORTHOGONALIZE_X_LATER

인 기본 설정만 사용했으므로 해당하지 않는 설정 코드는 스킵한다.

setting_solveMode = SOLVER_FIX_LAMBDA | SOLVER_ORTHOGONALIZE_X_LATER

<https://github.com/JakobEngel/dso/blob/master/src/OptimizationBackend/EnergyFunctional.cpp#L775>

4.3.1. Keyframes → Sliding Window Optimization

CODE REVIEW

EnergyFunctional.cpp::solveSystemF()

```

VecX x;
if(setting_solverMode & SOLVER_SVD)
{
    VecX SVecI = HFinal_top.diagonal().cwiseSqrt().cwiseInverse();
    MatXX HFinalScaled = SVecI.asDiagonal() * HFinal_top * SVecI.asDiagonal();
    VecX bFinalScaled = SVecI.asDiagonal() * bFinal_top;
    Eigen::JacobiSVD<MatXX> svd(HFinalScaled, Eigen::ComputeThinU | Eigen::ComputeThinV);
    VecX S = svd.singularValues();

    double minSv = 1e10, maxSv = 0;

    for(int i=0;i<S.size();i++)
    {
        if(S[i] < minSv) minSv = S[i];
        if(S[i] > maxSv) maxSv = S[i];
    }
    VecX Ub = svd.matrixU().transpose()*bFinalScaled;
    int setZero=0;
    for(int i=0;i<Ub.size();i++)
    {
        if(S[i] < setting_solverModeDelta*maxSv)
            { Ub[i] = 0; setZero++; }

        if((setting_solverMode & SOLVER_SVD_CUT7) && (i >= Ub.size()-7))
            { Ub[i] = 0; setZero++; }
        else Ub[i] /= S[i];
    }
    x = SVecI.asDiagonal() * svd.matrixV() * Ub;
}

else
{
    VecX SVecI = (HFinal_top.diagonal() + VecX::Constant(HFinal_top.cols(),
    10)).cwiseSqrt().cwiseInverse();
    MatXX HFinalScaled = SVecI.asDiagonal() * HFinal_top * SVecI.asDiagonal();
    x = SVecI.asDiagonal() * HFinalScaled.ldlt().solve(SVecI.asDiagonal() * bFinal_top);
}

```

다음을 순서대로 계산한다

$$\begin{aligned} \mathbf{s} &= \begin{bmatrix} \text{diag}(\mathbf{H}_{zz})_1 + 10 \\ \vdots \\ \text{diag}(\mathbf{H}_{zz})_9 + 10 \end{bmatrix} \\ \mathbf{s} &= \frac{1}{\sqrt{\mathbf{s}}} \end{aligned}$$

$$\mathbf{H}_{zz} = \text{diag}(\mathbf{s}) \cdot \mathbf{H}_{zz} \cdot \text{diag}(\mathbf{s})$$

결론적으로 Cholesky Decomposition을 사용하여 업데이트 값을 구한다

이 때, \mathbf{x} 값이 아닌 $-\mathbf{x}$ 값을 구한다는 것에 유의

$$\mathbf{x} = -\Delta \mathbf{x}_y = \text{diag}(\mathbf{s}) \mathbf{H}_{zz}^{-1} (\text{diag}(\mathbf{s}) \cdot \mathbf{b}_z)$$

$$\mathbf{y} = [\delta\xi^T \ a \ b \ c]$$

setting_solveMode = SOLVER_FIX_LAMBDA | SOLVER_ORTHOGONALIZE_X_LATER

<https://github.com/JakobEngel/dso/blob/master/src/OptimizationBackend/EnergyFunctional.cpp#L775>

4.3.1. Keyframes → Sliding Window Optimization

CODE REVIEW

`EnergyFunctional.cpp::solveSystemF()`

지금까지 Schur Complement로 $\Delta\mathbf{x}_y$ 를 구하는 과정을 수식으로 표현하면 다음과 같다

$$\begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho y} \\ \mathbf{H}_{y\rho} & \mathbf{H}_{yy} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{x}_\rho \\ \Delta\mathbf{x}_y \end{bmatrix} = \begin{bmatrix} \mathbf{b}_\rho \\ \mathbf{b}_y \end{bmatrix}$$

①

$$\Rightarrow \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ -\mathbf{H}_{y\rho}\mathbf{H}_{\rho\rho}^{-1} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho y} \\ \mathbf{H}_{y\rho} & \mathbf{H}_{yy} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{x}_\rho \\ \Delta\mathbf{x}_y \end{bmatrix} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ -\mathbf{H}_{y\rho}\mathbf{H}_{\rho\rho}^{-1} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{b}_\rho \\ \mathbf{b}_y \end{bmatrix}$$

②

$$\Rightarrow \begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho y} \\ \mathbf{0} & -\mathbf{H}_{y\rho}\mathbf{H}_{\rho\rho}^{-1}\mathbf{H}_{\rho y} + \mathbf{H}_{yy} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{x}_\rho \\ \Delta\mathbf{x}_y \end{bmatrix} = \begin{bmatrix} \mathbf{b}_\rho \\ -\mathbf{H}_{y\rho}\mathbf{H}_{\rho\rho}^{-1}\mathbf{b}_\rho + \mathbf{b}_y \end{bmatrix}$$

③

$$\Rightarrow (\mathbf{H}_{yy} - \mathbf{H}_{y\rho}\mathbf{H}_{\rho\rho}^{-1}\mathbf{H}_{\rho y})\Delta\mathbf{x}_y = (\mathbf{b}_y - \mathbf{b}_{sc})$$

$$\Rightarrow (\mathbf{H}_{yy} - \mathbf{H}_{sc})\Delta\mathbf{x}_y = (\mathbf{b}_y - \mathbf{b}_{sc})$$

④

$$-\Delta\mathbf{x}_y = (\mathbf{H}_{yy} - \mathbf{H}_{sc})^{-1}(\mathbf{b}_{sc} - \mathbf{b}_y)$$

$$= \mathbf{H}_{zz}^{-1}\mathbf{b}_z$$

-x 업데이트 값을 구한다는 것에 유의

$$\begin{aligned} \mathbf{y} &= [\delta\xi^T \ a \ b \ \mathbf{c}] \\ \mathbf{H}_{\rho\rho} &= \sum \mathbf{J}_\rho^T \mathbf{J}_\rho \\ \mathbf{H}_{\rho y} &= \sum \mathbf{J}_\rho^T \mathbf{J}_y \\ \mathbf{H}_{y\rho} &= \sum \mathbf{J}_y^T \mathbf{J}_\rho \\ \mathbf{H}_{yy} &= \sum \mathbf{J}_y^T \mathbf{J}_y \\ \mathbf{b}_\rho &= -\sum \mathbf{J}_\rho^T \mathbf{r}_{21} \\ \mathbf{b}_y &= -\sum \mathbf{J}_y^T \mathbf{r}_{21} \\ \mathbf{H}_{sc} &= \mathbf{H}_{y\rho}\mathbf{H}_{\rho\rho}^{-1}\mathbf{H}_{\rho y} \\ \mathbf{b}_{sc} &= \mathbf{H}_{y\rho}\mathbf{H}_{\rho\rho}^{-1}\mathbf{b}_\rho \end{aligned}$$

4.3.1. Keyframes → Sliding Window Optimization

EnergyFunctional.cpp::resubstituteF_MT()

```

void EnergyFunctional::resubstituteF_MT(VecX x, CalibHessian* HCalib, bool MT)
{
    assert(x.size() == CPARS+nFrames*8);

    VecXF xF = x.cast<float>();
    HCalib->step = -x.head<CPARS>();
    Mat18f* xAd = new Mat18f[nFrames*nFrames];
    VecCf cstep = xF.head<CPARS>();

    for(EFFrame* h : frames)
    {
        h->data->step.head<8>() = -x.segment<8>(CPARS+8*h->idx);
        h->data->step.tail<2>().setZero();

        for(EFFrame* t : frames)
            xAd[nFrames*h->idx + t->idx] = xF.segment<8>(CPARS+8*h->idx).transpose() * adHostF[h->idx+nFrames*t->idx]
                + xF.segment<8>(CPARS+8*t->idx).transpose() * adTargetF[h->idx+nFrames*t->idx];
    }

    if(MT)
        red->reduce(boost::bind(&EnergyFunctional::resubstituteFPt,
                               this, cstep, xAd, _1, _2, _3, _4), 0, allPoints.size(), 50);
    else
        resubstituteFPt(cstep, xAd, 0, allPoints.size(), 0, 0);

    delete[] xAd;
}

```

이전 단계에서 $-x$ 업데이트 값을 구한다는 것에 유의

$$-\Delta \mathbf{x}_y = \left[-\delta \mathbf{c} \quad -\sum_{i=1}^N \xi'(i) \right]$$

\mathbf{x}_F $-\Delta \mathbf{x}_y$ 업데이트 값 x 의 float 타입

$HCalib->step$ $\delta \mathbf{c}$ Camera Intrinsic 파라미터 업데이트 값

$cstep$ $-\delta \mathbf{c}$ Camera Intrinsic 파라미터 업데이트 값의 float 타입

$$\mathbf{y} = [\delta \xi^T \ a \ b \ \mathbf{c}]$$

$$\xi' = [\delta \xi \ a \ b]_{8 \times 1}$$

<https://github.com/JakobEngel/dso/blob/master/src/OptimizationBackend/EnergyFunctional.cpp#L263>

4.3.1. Keyframes → Sliding Window Optimization

CODE REVIEW

EnergyFunctional.cpp::resubstituteF_MT()

```

void EnergyFunctional::resubstituteF_MT(VecX x, CalibHessian* HCalib, bool MT)
{
    assert(x.size() == CPARS+nFrames*8);

    VecXF xF = x.cast<float>();
    HCalib->step = -x.head<CPARS>();
    Mat18f* xAd = new Mat18f[nFrames*nFrames];
    VecCf cstep = xF.head<CPARS>();

    for(EFFrame* h : frames)
    {
        h->data->step.head<8>() = -x.segment<8>(CPARS+8*h->idx);
        h->data->step.tail<2>().setZero();

        for(EFFrame* t : frames)
            xAd[nFrames*h->idx + t->idx] = xF.segment<8>(CPARS+8*h-
                >idx).transpose() * adHostF[h->idx+nFrames*t->idx]
                + xF.segment<8>(CPARS+8*t->idx).transpose() * adTargetF[h-
                >idx+nFrames*t->idx];
    }

    if(MT)
        red->reduce(boost::bind(&EnergyFunctional::resubstituteFPt,
            this, cstep, xAd, _1, _2, _3, _4), 0, allPoints.size(), 50);
    else
        resubstituteFPt(cstep, xAd, 0, allPoints.size(), 0, 0);

    delete[] xAd;
}

```

h->data->step

$$\xi'_1$$

host 키프레임의 업데이트 값 설정
(FullSystemOptimize.cpp::doStepFromBackup()에서 실제로 업데이트한다)

xAd

$$-\left(\frac{\partial \xi'_{21}}{\partial \xi'_1}\right)\xi'_1 - \left(\frac{\partial \xi'_{21}}{\partial \xi'_2}\right)\xi'_2$$

Adjoint 행렬을 구한다

1: host, 2: target

$$\left(\frac{\partial \xi'_{21}}{\partial \xi'_1}\right) = \begin{bmatrix} \left(\frac{\partial \xi_{21}}{\partial \xi_1}\right)^T & \left(\frac{\partial \mathbf{l}_{21}}{\partial \mathbf{l}_1}\right)^T \end{bmatrix}$$

$$\xi' = [\delta \xi \ a \ b]_{8 \times 1}$$

<https://github.com/JakobEngel/dso/blob/master/src/OptimizationBackend/EnergyFunctional.cpp#L263>

4.3.1. Keyframes → Sliding Window Optimization

CODE REVIEW

EnergyFunctional.cpp::resubstituteF_MT()

```
void EnergyFunctional::resubstituteF_MT(VecX x, CalibHessian* HCalib, bool MT)
{
    assert(x.size() == CPARS+nFrames*8);

    VecXF xF = x.cast<float>();
    HCalib->step = -x.head<CPARS>();
    Mat18f* xAd = new Mat18f[nFrames*nFrames];
    VecCf cstep = xF.head<CPARS>();

    for(EFFrame* h : frames)
    {
        h->data->step.head<8>() = -x.segment<8>(CPARS+8*h->idx);
        h->data->step.tail<2>().setZero();

        for(EFFrame* t : frames)
            xAd[nFrames*h->idx + t->idx] = xF.segment<8>(CPARS+8*h-
                >idx).transpose() * adHostF[h->idx+nFrames*t->idx]
                + xF.segment<8>(CPARS+8*t->idx).transpose() * adTargetF[h-
                >idx+nFrames*t->idx];
    }

    if(MT)
        red->reduce(boost::bind(&EnergyFunctional::resubstituteFPt,
            this, cstep, xAd, _1, _2, _3, _4), 0, allPoints.size(), 50);
    else
        resubstituteFPt(cstep, xAd, 0, allPoints.size(), 0, 0);

    delete[] xAd;
}
```

Inverse Depth 값 업데이트 후 xAd 제거

<https://github.com/JakobEngel/dso/blob/master/src/OptimizationBackend/EnergyFunctional.cpp#L263>

4.3.1. Keyframes → Sliding Window Optimization

CODE REVIEW

EnergyFunctional.cpp::resubstituteFPt()

```
void EnergyFunctional::resubstituteFPt(const VecCf &xc, Mat18f* xAd, int min, int max, Vec10* stats, int tid)
{
    for(int k=min;k<max;k++)
    {
        EFPPoint* p = allPoints[k];
        int ngoodres = 0;
        for(EFResidual* r : p->residualsAll) if(r->isActive()) ngoodres++;
        if(ngoodres==0)
        {
            p->data->step = 0;
            continue;
        }

        float b = p->bdSumF;
        b -= xc.dot(p->Hcd_accAF + p->Hcd_accLF);

        for(EFResidual* r : p->residualsAll)
        {
            if(!r->isActive()) continue;
            b -= xAd[r->hostIDX*nFrames + r->targetIDX] * r->JpJdF;
        }

        p->data->step = - b*p->HdiF;
        assert(std::isfinite(p->data->step));
    }
}
```

active 상태가 아닌 포인트는 업데이트 생략

<https://github.com/JakobEngel/dso/blob/master/src/OptimizationBackend/EnergyFunctional.cpp#L263>

4.3.1. Keyframes → Sliding Window Optimization

CODE REVIEW

EnergyFunctional.cpp::resubstituteFPt()

```

void EnergyFunctional::resubstituteFPt(const VecCf &xc, Mat18f* xAd, int min, int
max, Vec10* stats, int tid)
{
    for(int k=min;k<max;k++)
    {
        EFPPoint* p = allPoints[k];
        int ngoodres = 0;
        for(EFResidual* r : p->residualsAll) if(r->isActive()) ngoodres++;
        if(ngoodres==0)
        {
            p->data->step = 0;
            continue;
        }

        float b = p->bdSumF;
        b -= xc.dot(p->Hcd_accAF + p->Hcd_accLF);

        for(EFResidual* r : p->residualsAll)
        {
            if(!r->isActive()) continue;
            b -= xAd[r->hostIDX*nFrames + r->targetIDX] * r->JpJdF;
        }

        p->data->step = - b*p->HdiF;
        assert(std::isfinite(p->data->step));
    }
}

```

다음을 순서대로 계산한다

$$\mathbf{b}^{(j)} = \sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho^{(j)}}^T \mathbf{r}_{21}^{(i)}$$

$$\mathbf{b}^{(j)} = \mathbf{b}^{(j)} + \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \mathbf{c}}^T \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho^{(j)}} \right) \partial \mathbf{c}$$

$$\mathbf{b}^{(j)} = \mathbf{b}^{(j)} + \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \xi'_1}^T \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho^{(j)}} \right) \xi'_1 + \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \xi'_2}^T \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho^{(j)}} \right) \xi'_2$$

$\mathbf{p}->\text{data}->\text{step}$

$$\Delta \mathbf{x}_\rho^{(j)} = -(\mathbf{H}_{\rho\rho}^{(j)})^{-1} \mathbf{b}^{(j)}$$

$$\mathbf{y} = [\delta\xi^T \ a \ b \ \mathbf{c}]$$

$$\xi' = [\delta\xi \ a \ b]_{8 \times 1}$$

<https://github.com/JakobEngel/dso/blob/master/src/OptimizationBackend/EnergyFunctional.cpp#L263>

4.3.1. Keyframes → Sliding Window Optimization

CODE REVIEW

`EnergyFunctional.cpp::resubstituteFPt()`

지금까지 Schur Complement로 $\Delta \mathbf{x}_\rho$ 를 구하는 과정을 수식으로 표현하면 다음과 같다

$$\begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho\mathbf{y}} \\ \mathbf{H}_{\mathbf{y}\rho} & \mathbf{H}_{\mathbf{y}\mathbf{y}} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x}_\rho \\ \Delta \mathbf{x}_\mathbf{y} \end{bmatrix} = \begin{bmatrix} \mathbf{b}_\rho \\ \mathbf{b}_\mathbf{y} \end{bmatrix}$$

$$① \Rightarrow \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ -\mathbf{H}_{\mathbf{y}\rho} \mathbf{H}_{\rho\rho}^{-1} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho\mathbf{y}} \\ \mathbf{H}_{\mathbf{y}\rho} & \mathbf{H}_{\mathbf{y}\mathbf{y}} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x}_\rho \\ \Delta \mathbf{x}_\mathbf{y} \end{bmatrix} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ -\mathbf{H}_{\mathbf{y}\rho} \mathbf{H}_{\rho\rho}^{-1} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{b}_\rho \\ \mathbf{b}_\mathbf{y} \end{bmatrix}$$

$$② \Rightarrow \begin{bmatrix} \mathbf{H}_{\rho\rho} & \mathbf{H}_{\rho\mathbf{y}} \\ \mathbf{0} & -\mathbf{H}_{\mathbf{y}\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{H}_{\rho\mathbf{y}} + \mathbf{H}_{\mathbf{y}\mathbf{y}} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x}_\rho \\ \Delta \mathbf{x}_\mathbf{y} \end{bmatrix} = \begin{bmatrix} \mathbf{b}_\rho \\ -\mathbf{H}_{\mathbf{y}\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{b}_\rho + \mathbf{b}_\mathbf{y} \end{bmatrix}$$

$$\begin{aligned} &\Rightarrow \mathbf{H}_{\rho\rho} \Delta \mathbf{x}_\rho + \mathbf{H}_{\rho\mathbf{y}} \Delta \mathbf{x}_\mathbf{y} = \mathbf{b}_\rho \\ &\Rightarrow \Delta \mathbf{x}_\rho = \mathbf{H}_{\rho\rho}^{-1} (\mathbf{b}_\rho - \mathbf{H}_{\rho\mathbf{y}} \Delta \mathbf{x}_\mathbf{y}) \\ &\Rightarrow \Delta \mathbf{x}_\rho = - \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho}^T \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho} \right)^{-1} \left(\sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho}^T \mathbf{r}_{21}^{(i)} + \sum_{i=1}^N \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \rho}^T \left(\frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \mathbf{c}} \partial \mathbf{c} + \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \xi'_1} \xi'_1 + \frac{\partial \mathbf{r}_{21}^{(i)}}{\partial \xi'_2} \xi'_2 \right) \right) \end{aligned}$$

$\Delta \mathbf{x}_\mathbf{y}$ 는 이미 알고 있는 값이다

$$\begin{aligned} \mathbf{y} &= [\delta \xi^T \ a \ b \ \mathbf{c}] \\ \mathbf{H}_{\rho\rho} &= \sum \mathbf{J}_\rho^T \mathbf{J}_\rho \\ \mathbf{H}_{\rho\mathbf{y}} &= \sum \mathbf{J}_\rho^T \mathbf{J}_\mathbf{y} \\ \mathbf{H}_{\mathbf{y}\rho} &= \sum \mathbf{J}_\mathbf{y}^T \mathbf{J}_\rho \\ \mathbf{H}_{\mathbf{y}\mathbf{y}} &= \sum \mathbf{J}_\mathbf{y}^T \mathbf{J}_\mathbf{y} \\ \mathbf{b}_\rho &= - \sum \mathbf{J}_\rho^T \mathbf{r}_{21} \\ \mathbf{b}_\mathbf{y} &= - \sum \mathbf{J}_\mathbf{y}^T \mathbf{r}_{21} \\ \mathbf{H}_{sc} &= \mathbf{H}_{\mathbf{y}\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{H}_{\rho\mathbf{y}} \\ \mathbf{b}_{sc} &= \mathbf{H}_{\mathbf{y}\rho} \mathbf{H}_{\rho\rho}^{-1} \mathbf{b}_\rho \end{aligned}$$

4.4. Null Space Effect Elimination

CODE REVIEW

HessianBlocks.cpp::setStateZero()::L74

```
void FrameHessian::setStateZero(const Vec10 &state_zero)
{
    assert(state_zero.head<6>().squaredNorm() < 1e-20);
    this->state_zero = state_zero;
    for(int i=0;i<6;i++)
    {
        Vec6 eps; eps.setZero(); eps[i] = 1e-3;
        SE3 EepsP = Sophus::SE3::exp(eps);
        SE3 EepsM = Sophus::SE3::exp(-eps);
        SE3 w2c_leftEps_P_x0 = (get_worldToCam_evalPT() * EepsP) * get_worldToCam_evalPT().inverse();
        SE3 w2c_leftEps_M_x0 = (get_worldToCam_evalPT() * EepsM) * get_worldToCam_evalPT().inverse();
        nullspaces_pose.col(i) = (w2c_leftEps_P_x0.log() - w2c_leftEps_M_x0.log())/(2e-3);
    }

    // scale change
    SE3 w2c_leftEps_P_x0 = (get_worldToCam_evalPT());
    w2c_leftEps_P_x0.translation() *= 1.00001;
    w2c_leftEps_P_x0 = w2c_leftEps_P_x0 * get_worldToCam_evalPT().inverse();
    SE3 w2c_leftEps_M_x0 = (get_worldToCam_evalPT());
    w2c_leftEps_M_x0.translation() /= 1.00001;
    w2c_leftEps_M_x0 = w2c_leftEps_M_x0 * get_worldToCam_evalPT().inverse();
    nullspaces_scale = (w2c_leftEps_P_x0.log() - w2c_leftEps_M_x0.log())/(2e-3);

    nullspaces_affine.setZero();
    nullspaces_affine.topLeftCorner<2,1>() = Vec2(1,0);
    assert(ab_exposure > 0);
    nullspaces_affine.topRightCorner<2,1>() = Vec2(0, expf(aff_g2l_0().a)*ab_exposure);
};


```

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/HessianBlocks.cpp#L74>

4.4. Null Space Effect Elimination

CODE REVIEW

HessianBlocks.cpp::setStateZero()::L74

```

void FrameHessian::setStateZero(const Vec10 &state_zero)
{
    assert(state_zero.head<6>().squaredNorm() < 1e-20);
    this->state_zero = state_zero;
    for(int i=0;i<6;i++)
    {
        Vec6 eps; eps.setZero(); eps[i] = 1e-3;
        SE3 EepsP = Sophus::SE3::exp(eps);
        SE3 EepsM = Sophus::SE3::exp(-eps);
        SE3 w2c_leftEps_P_x0 = (get_worldToCam_evalPT() * EepsP) * get_worldToCam_evalPT().inverse();
        SE3 w2c_leftEps_M_x0 = (get_worldToCam_evalPT() * EepsM) * get_worldToCam_evalPT().inverse();
        nullspaces_pose.col(i) = (w2c_leftEps_P_x0.log() - w2c_leftEps_M_x0.log())/(2e-3);
    }

    // scale change
    SE3 w2c_leftEps_P_x0 = (get_worldToCam_evalPT());
    w2c_leftEps_P_x0.translation() *= 1.00001;
    w2c_leftEps_P_x0 = w2c_leftEps_P_x0 * get_worldToCam_evalPT().inverse();
    SE3 w2c_leftEps_M_x0 = (get_worldToCam_evalPT());
    w2c_leftEps_M_x0.translation() /= 1.00001;
    w2c_leftEps_M_x0 = w2c_leftEps_M_x0 * get_worldToCam_evalPT().inverse();
    nullspaces_scale = (w2c_leftEps_P_x0.log() - w2c_leftEps_M_x0.log())/(2e-3);

    nullspaces_affine.setZero();
    nullspaces_affine.topLeftCorner<2,1>() = Vec2(1,0);
    assert(ab_exposure > 0);
    nullspaces_affine.topRightCorner<2,1>() = Vec2(0, expf(aff_g2l_0().a)*ab_exposure);
}

```

Pose에 대한 Null Space를 구하는 코드

$$\mathbf{T}_i^+ = \mathbf{T}_{cw} \exp(\boldsymbol{\epsilon}_i^+) \mathbf{T}_{cw}^{-1}$$

$$\mathbf{T}_i^- = \mathbf{T}_{cw} \exp(\boldsymbol{\epsilon}_i^-) \mathbf{T}_{cw}^{-1}$$

$$\mathbf{N}_{pose,i} = \frac{\log(\mathbf{T}_i^+) - \log(\mathbf{T}_i^-)}{2\epsilon} \quad i=1, \dots, 6$$

$\boldsymbol{\epsilon}_i^\pm$: i번째 열이 +-epsilon 값을 가지는 벡터

$$\boldsymbol{\epsilon}_1^+ = [\epsilon \ 0 \ 0 \ 0 \ 0 \ 0]^T$$

$$\boldsymbol{\epsilon}_6^+ = [0 \ 0 \ 0 \ 0 \ 0 \ \epsilon]^T$$

4.4. Null Space Effect Elimination

CODE REVIEW

HessianBlocks.cpp::setStateZero()::L74

```

void FrameHessian::setStateZero(const Vec10 &state_zero)
{
    assert(state_zero.head<6>().squaredNorm() < 1e-20);
    this->state_zero = state_zero;
    for(int i=0;i<6;i++)
    {
        Vec6 eps; eps.setZero(); eps[i] = 1e-3;
        SE3 EepsP = Sophus::SE3::exp(eps);
        SE3 EepsM = Sophus::SE3::exp(-eps);
        SE3 w2c_leftEps_P_x0 = (get_worldToCam_evalPT() * EepsP) * get_worldToCam_evalPT().inverse();
        SE3 w2c_leftEps_M_x0 = (get_worldToCam_evalPT() * EepsM) * get_worldToCam_evalPT().inverse();
        nullspaces_pose.col(i) = (w2c_leftEps_P_x0.log() - w2c_leftEps_M_x0.log())/(2e-3);
    }
}

// scale change
SE3 w2c_leftEps_P_x0 = (get_worldToCam_evalPT());
w2c_leftEps_P_x0.translation() *= 1.00001;
w2c_leftEps_P_x0 = w2c_leftEps_P_x0 * get_worldToCam_evalPT().inverse();
SE3 w2c_leftEps_M_x0 = (get_worldToCam_evalPT());
w2c_leftEps_M_x0.translation() /= 1.00001;
w2c_leftEps_M_x0 = w2c_leftEps_M_x0 * get_worldToCam_evalPT().inverse();
nullspaces_scale = (w2c_leftEps_P_x0.log() - w2c_leftEps_M_x0.log())/(2e-3);

nullspaces_affine.setZero();
nullspaces_affine.topLeftCorner<2,1>() = Vec2(1,0);
assert(ab_exposure > 0);
nullspaces_affine.topRightCorner<2,1>() = Vec2(0, expf(aff_g2l_0().a)*ab_exposure);
};

```

Scale에 대한 Null Space를 구하는 코드

$$\mathbf{T}^+ = \begin{bmatrix} \mathbf{R}_{cw} & 1.00001\mathbf{t}_{cw} \\ \mathbf{0} & 1 \end{bmatrix} \cdot \mathbf{T}_{cw}^{-1}$$

$$\mathbf{T}^- = \begin{bmatrix} \mathbf{R}_{cw} & -1.00001\mathbf{t}_{cw} \\ \mathbf{0} & 1 \end{bmatrix} \cdot \mathbf{T}_{cw}^{-1}$$

$$\mathbf{N}_{scale} = \frac{\log(\mathbf{T}^+) - \log(\mathbf{T}^-)}{2\epsilon}$$

4.4. Null Space Effect Elimination

HessianBlocks.cpp::setStateZero()::L74

```

void FrameHessian::setStateZero(const Vec10 &state_zero)
{
    assert(state_zero.head<6>().squaredNorm() < 1e-20);
    this->state_zero = state_zero;
    for(int i=0;i<6;i++)
    {
        Vec6 eps; eps.setZero(); eps[i] = 1e-3;
        SE3 EepsP = Sophus::SE3::exp(eps);
        SE3 EepsM = Sophus::SE3::exp(-eps);
        SE3 w2c_leftEps_P_x0 = (get_worldToCam_evalPT() * EepsP) * get_worldToCam_evalPT().inverse();
        SE3 w2c_leftEps_M_x0 = (get_worldToCam_evalPT() * EepsM) * get_worldToCam_evalPT().inverse();
        nullspaces_pose.col(i) = (w2c_leftEps_P_x0.log() - w2c_leftEps_M_x0.log())/(2e-3);
    }

    // scale change
    SE3 w2c_leftEps_P_x0 = (get_worldToCam_evalPT());
    w2c_leftEps_P_x0.translation() *= 1.00001;
    w2c_leftEps_P_x0 = w2c_leftEps_P_x0 * get_worldToCam_evalPT().inverse();
    SE3 w2c_leftEps_M_x0 = (get_worldToCam_evalPT());
    w2c_leftEps_M_x0.translation() /= 1.00001;
    w2c_leftEps_M_x0 = w2c_leftEps_M_x0 * get_worldToCam_evalPT().inverse();
    nullspaces_scale = (w2c_leftEps_P_x0.log() - w2c_leftEps_M_x0.log())/(2e-3);

    nullspaces_affine.setZero();
    nullspaces_affine.topLeftCorner<2,1>() = Vec2(1,0);
    assert(ab_exposure > 0);
    nullspaces_affine.topRightCorner<2,1>() = Vec2(0, expf(aff_g2l_0().a)*ab_exposure);
};

```

Photometric Param(a,b)에 대한 Null Space를 구하는 코드

→ orthogonalize() 코드를 보면 사용하지 않는 듯 하다

$$\mathbf{N}_{aff} = \begin{bmatrix} 1 & 0 \\ 0 & t \exp^a \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

4.4. Null Space Effect Elimination

CODE REVIEW

EnergyFunctional.cpp::orthogonalize():L718

```
void EnergyFunctional::orthogonalize(VecX* b, MatXX* H)
{
    // decide to which nullspaces to orthogonalize.
    std::vector<VecX> ns;
    ns.insert(ns.end(), lastNullspaces_pose.begin(), lastNullspaces_pose.end());
    ns.insert(ns.end(), lastNullspaces_scale.begin(), lastNullspaces_scale.end());

    // make Nullspaces matrix
    MatXX N(ns[0].rows(), ns.size());
    for(unsigned int i=0;i<ns.size();i++)
        N.col(i) = ns[i].normalized();

    // compute Npi := N * (N' * N)^-1 = pseudo inverse of N.
    Eigen::JacobiSVD<MatXX> svdNN(N, Eigen::ComputeThinU | Eigen::ComputeThinV);
    VecX SNN = svdNN.singularValues();
    double minSv = 1e10, maxSv = 0;

    for(int i=0;i<SNN.size();i++) {
        if(SNN[i] < minSv) minSv = SNN[i];
        if(SNN[i] > maxSv) maxSv = SNN[i];
    }

    for(int i=0;i<SNN.size();i++)
    { if(SNN[i] > setting_solverModeDelta*maxSv) SNN[i] = 1.0 / SNN[i]; else SNN[i] = 0; }

    MatXX Npi = svdNN.matrixU() * SNN.asDiagonal() * svdNN.matrixV().transpose(); // [dim] x 9.
    MatXX NNpiT = N*Npi.transpose(); // [dim] x [dim].
    MatXX NNpiTS = 0.5*(NNpiT + NNpiT.transpose()); // = N * (N' * N)^-1 * N'.

    if(b!=0) *b -= NNpiTS * *b;
    if(H!=0) *H -= NNpiTS * *H * NNpiTS;
}
```

<https://github.com/JakobEngel/dso/blob/master/src/OptimizationBackend/EnergyFunctional.cpp#L718>

4.4. Null Space Effect Elimination

CODE REVIEW

EnergyFunctional.cpp::orthogonalize()::L718

```
void EnergyFunctional::orthogonalize(VecX* b, MatXX* H)
{
    // decide to which nullspaces to orthogonalize.
    std::vector<VecX> ns;
    ns.insert(ns.end(), lastNullspaces_pose.begin(), lastNullspaces_pose.end());
    ns.insert(ns.end(), lastNullspaces_scale.begin(), lastNullspaces_scale.end());

    // make Nullspaces matrix
    MatXX N(ns[0].rows(), ns.size());
    for(unsigned int i=0;i<ns.size();i++)
        N.col(i) = ns[i].normalized();

    // compute Npi := N * (N' * N)^-1 = pseudo inverse of N.
    Eigen::JacobiSVD<MatXX> svdNN(N, Eigen::ComputeThinU | Eigen::ComputeThinV);
    VecX SNN = svdNN.singularValues();
    double minSv = 1e10, maxSv = 0;

    for(int i=0;i<SNN.size();i++) {
        if(SNN[i] < minSv) minSv = SNN[i];
        if(SNN[i] > maxSv) maxSv = SNN[i];
    }

    for(int i=0;i<SNN.size();i++)
    { if(SNN[i] > setting_solverModeDelta*maxSv) SNN[i] = 1.0 / SNN[i]; else SNN[i] = 0; }

    MatXX Npi = svdNN.matrixU() * SNN.asDiagonal() * svdNN.matrixV().transpose(); // [dim] x 9.
    MatXX NNpiT = N*Npi.transpose(); // [dim] x [dim].
    MatXX NNpiTS = 0.5*(NNpiT + NNpiT.transpose()); // = N * (N' * N)^-1 * N'.

    if(b!=0) *b -= NNpiTS * *b;
    if(H!=0) *H -= NNpiTS * *H * NNpiTS;
}
```

현재 윈도우 상에 있는 모든 키프레임들의 가장 최근 null space 행렬을 생성한다
이 때 각 열(Column)들을 정규화해준다

<https://github.com/JakobEngel/dso/blob/master/src/OptimizationBackend/EnergyFunctional.cpp#L718>

4.4. Null Space Effect Elimination

CODE REVIEW

EnergyFunctional.cpp::orthogonalize():L718

```
void EnergyFunctional::orthogonalize(VecX* b, MatXX* H)
{
    // decide to which nullspaces to orthogonalize.
    std::vector<VecX> ns;
    ns.insert(ns.end(), lastNullspaces_pose.begin(), lastNullspaces_pose.end());
    ns.insert(ns.end(), lastNullspaces_scale.begin(), lastNullspaces_scale.end());

    // make Nullspaces matrix
    MatXX N(ns[0].rows(), ns.size());
    for(unsigned int i=0;i<ns.size();i++)
        N.col(i) = ns[i].normalized();

    // compute Npi := N * (N' * N)^-1 = pseudo inverse of N.
    Eigen::JacobiSVD<MatXX> svdNN(N, Eigen::ComputeThinU | Eigen::ComputeThinV);
    VecX SNN = svdNN.singularValues();
    double minSv = 1e10, maxSv = 0;

    for(int i=0;i<SNN.size();i++) {
        if(SNN[i] < minSv) minSv = SNN[i];
        if(SNN[i] > maxSv) maxSv = SNN[i];
    }

    for(int i=0;i<SNN.size();i++)
    { if(SNN[i] > setting_solverModeDelta*maxSv) SNN[i] = 1.0 / SNN[i]; else SNN[i] = 0; }

    MatXX Npi = svdNN.matrixU() * SNN.asDiagonal() * svdNN.matrixV().transpose(); // [dim] x 9.
    MatXX NNpiT = N*Npi.transpose(); // [dim] x [dim].
    MatXX NNpiTS = 0.5*(NNpiT + NNpiT.transpose()); // = N * (N' * N)^-1 * N'.

    if(b!=0) *b -= NNpiTS * *b;
    if(H!=0) *H -= NNpiTS * *H * NNpiTS;
}
```

$\mathbf{N}^T \mathbf{N}$ 이 역행렬이 존재하는 것과 관계없이 SVD를 통해 \mathbf{N} 을 분해한다

$$\mathbf{N} = \mathbf{U}\mathbf{D}\mathbf{V}^T$$

<https://github.com/JakobEngel/dso/blob/master/src/OptimizationBackend/EnergyFunctional.cpp#L718>

4.4. Null Space Effect Elimination

CODE REVIEW

EnergyFunctional.cpp::orthogonalize():L718

```
void EnergyFunctional::orthogonalize(VecX* b, MatXX* H)
{
    // decide to which nullspaces to orthogonalize.
    std::vector<VecX> ns;
    ns.insert(ns.end(), lastNullspaces_pose.begin(), lastNullspaces_pose.end());
    ns.insert(ns.end(), lastNullspaces_scale.begin(), lastNullspaces_scale.end());

    // make Nullspaces matrix
    MatXX N(ns[0].rows(), ns.size());
    for(unsigned int i=0;i<ns.size();i++)
        N.col(i) = ns[i].normalized();

    // compute Npi := N * (N' * N)^-1 = pseudo inverse of N.
    Eigen::JacobiSVD<MatXX> svdNN(N, Eigen::ComputeThinU | Eigen::ComputeThinV);
    VecX SNN = svdNN.singularValues();
    double minSv = 1e10, maxSv = 0;

    for(int i=0;i<SNN.size();i++)
    {
        if(SNN[i] < minSv) minSv = SNN[i];
        if(SNN[i] > maxSv) maxSv = SNN[i];
    }

    for(int i=0;i<SNN.size();i++)
    { if(SNN[i] > setting_solverModeDelta*maxSv) SNN[i] = 1.0 / SNN[i]; else SNN[i] = 0; }

    MatXX Npi = svdNN.matrixU() * SNN.asDiagonal() * svdNN.matrixV().transpose(); // [dim] x 9.
    MatXX NNpiT = N*Npi.transpose(); // [dim] x [dim].
    MatXX NNpiTS = 0.5*(NNpiT + NNpiT.transpose()); // = N * (N' * N)^-1 * N'.

    if(b!=0) *b -= NNpiTS * *b;
    if(H!=0) *H -= NNpiTS * *H * NNpiTS;
}
```

Singular Value를 포함한 대각행렬 D 의 역행렬 D^\dagger 을 구하는 코드
→ 특정 threshold보다 작은 Singular Value들은 0으로 처리하고 큰 값들은 역수를 취함으로써 역행렬을 구한다

<https://github.com/JakobEngel/dso/blob/master/src/OptimizationBackend/EnergyFunctional.cpp#L718>

4.4. Null Space Effect Elimination

CODE REVIEW

EnergyFunctional.cpp::orthogonalize()::L718

```
void EnergyFunctional::orthogonalize(VecX* b, MatXX* H)
{
    // decide to which nullspaces to orthogonalize.
    std::vector<VecX> ns;
    ns.insert(ns.end(), lastNullspaces_pose.begin(), lastNullspaces_pose.end());
    ns.insert(ns.end(), lastNullspaces_scale.begin(), lastNullspaces_scale.end());

    // make Nullspaces matrix
    MatXX N(ns[0].rows(), ns.size());
    for(unsigned int i=0;i<ns.size();i++)
        N.col(i) = ns[i].normalized();

    // compute Npi := N * (N' * N)^-1 = pseudo inverse of N.
    Eigen::JacobiSVD<MatXX> svdNN(N, Eigen::ComputeThinU | Eigen::ComputeThinV);
    VecX SNN = svdNN.singularValues();
    double minSv = 1e10, maxSv = 0;

    for(int i=0;i<SNN.size();i++) {
        if(SNN[i] < minSv) minSv = SNN[i];
        if(SNN[i] > maxSv) maxSv = SNN[i];
    }

    for(int i=0;i<SNN.size();i++)
    { if(SNN[i] > setting_solverModeDelta*maxSv) SNN[i] = 1.0 / SNN[i]; else SNN[i] = 0; }

    MatXX Npi = svdNN.matrixU() * SNN.asDiagonal() * svdNN.matrixV().transpose(); // [dim] x 9.
    MatXX NNpiT = N*Npi.transpose(); // [dim] x [dim].
    MatXX NNpiTS = 0.5*(NNpiT + NNpiT.transpose()); // = N * (N' * N)^-1 * N'.

    if(b!=0) *b -= NNpiTS * *b;
    if(H!=0) *H -= NNpiTS * *H * NNpiTS;
}
```

다음을 순서대로 계산한다

(마지막 $\mathbf{NN}^{\dagger'}$ 를 왜 계산해야 하는지는 정확하게 이해하지 못함)

$$\mathbf{N}^{\dagger} = \mathbf{V}\mathbf{D}^{\dagger}\mathbf{U}^T$$

$$\mathbf{NN}^{\dagger} = \mathbf{N}\mathbf{V}\mathbf{D}^{\dagger}\mathbf{U}^T = \mathbf{NN}^{\dagger}$$

$$\mathbf{NN}^{\dagger'} = \frac{\mathbf{NN}^{\dagger} + (\mathbf{NN}^{\dagger})^T}{2}$$

4.4. Null Space Effect Elimination

CODE REVIEW

EnergyFunctional.cpp::orthogonalize():L718

```
void EnergyFunctional::orthogonalize(VecX* b, MatXX* H)
{
    // decide to which nullspaces to orthogonalize.
    std::vector<VecX> ns;
    ns.insert(ns.end(), lastNullspaces_pose.begin(), lastNullspaces_pose.end());
    ns.insert(ns.end(), lastNullspaces_scale.begin(), lastNullspaces_scale.end());

    // make Nullspaces matrix
    MatXX N(ns[0].rows(), ns.size());
    for(unsigned int i=0;i<ns.size();i++)
        N.col(i) = ns[i].normalized();

    // compute Npi := N * (N' * N)^-1 = pseudo inverse of N.
    Eigen::JacobiSVD<MatXX> svdNN(N, Eigen::ComputeThinU | Eigen::ComputeThinV);
    VecX SNN = svdNN.singularValues();
    double minSv = 1e10, maxSv = 0;

    for(int i=0;i<SNN.size();i++) {
        if(SNN[i] < minSv) minSv = SNN[i];
        if(SNN[i] > maxSv) maxSv = SNN[i];
    }

    for(int i=0;i<SNN.size();i++)
    { if(SNN[i] > setting_solverModeDelta*maxSv) SNN[i] = 1.0 / SNN[i]; else SNN[i] = 0; }

    MatXX Npi = svdNN.matrixU() * SNN.asDiagonal() * svdNN.matrixV().transpose(); // [dim] x 9.
    MatXX NNpiT = N*Npi.transpose(); // [dim] x [dim].
    MatXX NNpiTS = 0.5*(NNpiT + NNpiT.transpose()); // = N * (N' * N)^-1 * N'.

    if(b!=0) *b -= NNpiTS * *b;
    if(H!=0) *H -= NNpiTS * *H * NNpiTS;
}
```

해당 연산을 통해 Null Space의 영향을 제거한다

$$\mathbf{b}^* = \mathbf{b} - (\mathbf{N}\mathbf{N}^\dagger)^T \mathbf{b}$$

$$\mathbf{H}^* = \mathbf{H} - (\mathbf{N}\mathbf{N}^\dagger)^T \mathbf{H}(\mathbf{N}\mathbf{N}^\dagger)$$

5. Pixel Selection

CODE REVIEW

PixelSelector2.cpp::makeMaps()::L142

```
int PixelSelector::makeMaps(const FrameHessian* const fh, float* map_out, float density, int recursionsLeft, bool plot, float thFactor)
{
    float numHave=0;
    float numWant=density;
    float quotia;
    int idealPotential = currentPotential;
    if(fh != gradHistFrame) makeHists(fh);
    // select!
    Eigen::Vector3i n = this->select(fh, map_out, currentPotential, thFactor);
    // sub-select!
    numHave = n[0]+n[1]+n[2];
    quotia = numWant / numHave;
    // by default we want to over-sample by 40% just to be sure.
    float K = numHave * (currentPotential+1) * (currentPotential+1);
    idealPotential = sqrtf(K/numWant)-1; // round down.
    if(idealPotential<1) idealPotential=1;
    if( recursionsLeft>0 && quotia > 1.25 && currentPotential>1)
    {
        if(idealPotential>=currentPotential)
            idealPotential = currentPotential-1;
        currentPotential = idealPotential;
        return makeMaps(fh, map_out, density, recursionsLeft-1, plot, thFactor);
    }
    else if(recursionsLeft>0 && quotia < 0.25)
    {
        if(idealPotential<=currentPotential)
            idealPotential = currentPotential+1;
        currentPotential = idealPotential;
        return makeMaps(fh, map_out, density, recursionsLeft-1, plot, thFactor);
    }
}
```

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/PixelSelector2.cpp#L142>

5. Pixel Selection

CODE REVIEW

PixelSelector2.cpp::makeMaps()::L142

```
int PixelSelector::makeMaps(const FrameHessian* const fh, float* map_out, float density, int recursionsLeft, bool plot, float thFactor)
{
    float numHave=0;
    float numWant=density;
    float quotia;
    int idealPotential = currentPotential;
    if(fh != gradHistFrame) makeHists(fh);
    // select!
    Eigen::Vector3i n = this->select(fh, map_out, currentPotential, thFactor);
    // sub-select!
    numHave = n[0]+n[1]+n[2];
    quotia = numWant / numHave;
    // by default we want to over-sample by 40% just to be sure.
    float K = numHave * (currentPotential+1) * (currentPotential+1);
    idealPotential = sqrtf(K/numWant)-1; // round down.
    if(idealPotential<1) idealPotential=1;
    if( recursionsLeft>0 && quotia > 1.25 && currentPotential>1)
    {
        if(idealPotential>=currentPotential)
            idealPotential = currentPotential-1;
        currentPotential = idealPotential;
        return makeMaps(fh, map_out, density, recursionsLeft-1, plot,thFactor);
    }
    else if(recursionsLeft>0 && quotia < 0.25)
    {
        if(idealPotential<=currentPotential)
            idealPotential = currentPotential+1;
        currentPotential = idealPotential;
        return makeMaps(fh, map_out, density, recursionsLeft-1, plot,thFactor);
    }
}
```

Gradient Histogram + Dynamic Grid 방법을 사용하여 한 이미지 내에서 Pixel map을 계산하는 함수

► Gradient Histogram

► Dynamic Grid

5. Pixel Selection

CODE REVIEW

PixelSelector2.cpp::makeMaps()::L142

```
int PixelSelector::makeMaps(const FrameHessian* const fh, float* map_out, float density, int recursionsLeft, bool plot, float thFactor)
{
    float numHave=0;
    float numWant=density;
    float quotia;
    int idealPotential = currentPotential;
    if(fh != gradHistFrame) makeHists(fh);
    // select!
    Eigen::Vector3i n = this->select(fh, map_out, currentPotential, thFactor);
    // sub-select!
    numHave = n[0]+n[1]+n[2]; 각 피라미드 레벨별로 추출된 포인트의 개수를 합하고
    quotia = numWant / numHave; 원하는 포인트의 총량과 비율을 구한다.
    // by default we want to over-sample by 40% just to be sure.
    float K = numHave * (currentPotential+1) * (currentPotential+1);
    idealPotential = sqrtf(K/numWant)-1; // round down.
    if(idealPotential<1) idealPotential=1;
    if( recursionsLeft>0 && quotia > 1.25 && currentPotential>1)
    {
        if(idealPotential>=currentPotential)
            idealPotential = currentPotential-1;
        currentPotential = idealPotential;
        return makeMaps(fh, map_out, density, recursionsLeft-1, plot, thFactor);
    }
    else if(recursionsLeft>0 && quotia < 0.25)
    {
        if(idealPotential<=currentPotential)
            idealPotential = currentPotential+1;
        currentPotential = idealPotential;
        return makeMaps(fh, map_out, density, recursionsLeft-1, plot, thFactor);
    }
}
```

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/PixelSelector2.cpp#L142>

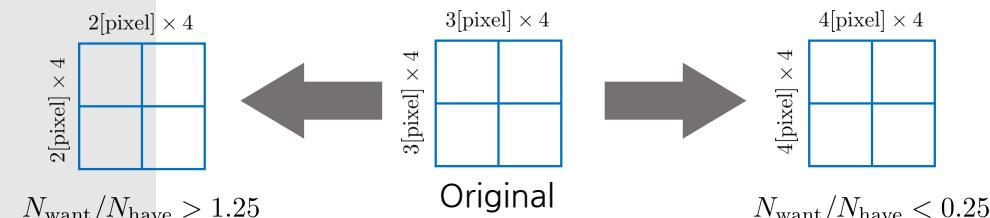
5. Pixel Selection

CODE REVIEW

PixelSelector2.cpp::makeMaps()::L142

```
int PixelSelector::makeMaps(const FrameHessian* const fh, float* map_out, float density, int recursionsLeft, bool plot, float thFactor)
{
    float numHave=0;
    float numWant=density;
    float quotia;
    int idealPotential = currentPotential;
    if(fh != gradHistFrame) makeHists(fh);
    // select!
    Eigen::Vector3i n = this->select(fh, map_out, currentPotential, thFactor);
    // sub-select!
    numHave = n[0]+n[1]+n[2];
    quotia = numWant / numHave;
    // by default we want to over-sample by 40% just to be sure.
    float K = numHave * (currentPotential+1) * (currentPotential+1);
    idealPotential = sqrtf(K/numWant)-1; // round down.
    if(idealPotential<1) idealPotential=1;
    if( recursionsLeft>0 && quotia > 1.25 && currentPotential>1)
    {
        if(idealPotential>=currentPotential)
            idealPotential = currentPotential-1;
        currentPotential = idealPotential;
        return makeMaps(fh, map_out, density, recursionsLeft-1, plot,thFactor);
    }
    else if(recursionsLeft>0 && quotia < 0.25)
    {
        if(idealPotential<=currentPotential)
            idealPotential = currentPotential+1;
        currentPotential = idealPotential;
        return makeMaps(fh, map_out, density, recursionsLeft-1, plot,thFactor);
    }
}
```

추출된 포인트의 개수와 원하는 포인트의 총량의 비율을 비교하여 너무 많이 추출되거나 너무 적게 추출되는 경우 Grid의 크기를 조절하고 다시 포인트를 추출한다.



5. Pixel Selection

CODE REVIEW

PixelSelector2.cpp::makeHists():L78

```
float * mapmax0 = fh->absSquaredGrad[0];
int w = wG[0];
int h = hG[0];
int w32 = w/32;
int h32 = h/32;
thsStep = w32;
for(int y=0;y<h32;y++) {
    for(int x=0;x<w32;x++) {
        float* map0 = mapmax0+32*x+32*y*w;
        int* hist0 = gradHist;// + 50*(x+y*w32);
        memset(hist0,0,sizeof(int)*50);
        for(int j=0;j<32;j++) for(int i=0;i<32;i++){
            int it = i+32*x;
            int jt = j+32*y;
            if(it>w-2 || jt>h-2 || it<1 || jt<1) continue;
            int g = sqrtf(map0[i+j*w]);
            if(g>48) g=48;
            hist0[g+1]++;
            hist0[0]++;
        }
        ths[x+y*w32] = computeHistQuantil(hist0,setting_minGradHistCut) + setting_minGradHistAdd;
    }
    for(int y=0;y<h32;y++) {
        for(int x=0;x<w32;x++) {
            float sum=0,num=0;
            if(x>0)
            {
                if(y>0) {num++; sum+=ths[x-1+(y-1)*w32];}
                if(y<h32-1) {num++; sum+=ths[x-1+(y+1)*w32];}
                num++; sum+=ths[x-1+(y)*w32];
            }
            if(x<w32-1)
            {
                if(y>0) {num++; sum+=ths[x+1+(y-1)*w32];}
                if(y<h32-1) {num++; sum+=ths[x+1+(y+1)*w32];}
                num++; sum+=ths[x+1+(y)*w32];
            }
            if(y>0) {num++; sum+=ths[x+(y-1)*w32];}
            if(y<h32-1) {num++; sum+=ths[x+(y+1)*w32];}
            num++; sum+=ths[x+y*w32];
            thsSmoothed[x+y*w32] = (sum/num) * (sum/num);
        }
    }
}
```

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/PixelSelector2.cpp#L78>

5. Pixel Selection

CODE REVIEW

PixelSelector2.cpp::makeHists()::L78

```
float * mapmax0 = fh->absSquaredGrad[0];
int w = wG[0];
int h = hG[0];
int w32 = w/32;
int h32 = h/32;
thsStep = w32;
for(int y=0;y<h32;y++)
for(int x=0;x<w32;x++) {
    float* map0 = mapmax0+32*x+32*y*w;
    int* hist0 = gradHist;// + 50*(x+y*w32);
    memset(hist0,0,sizeof(int)*50);
    for(int j=0;j<32;j++) for(int i=0;i<32;i++){
        int it = i+32*x;
        int jt = j+32*y;
        if(it>w-2 || jt>h-2 || it<1 || jt<1) continue;
        int g = sqrtf(map0[i+j*w]);
        if(g>48) g=48;
        hist0[g+1]++;
        hist0[0]++;
    }
    ths[x+y*w32] = computeHistQuantil(hist0,setting_minGradHistCut) + setting_minGradHistAdd;
}
for(int y=0;y<h32;y++)
for(int x=0;x<w32;x++) {
    float sum=0,num=0;
    if(x>0)
    {
        if(y>0) {num++; sum+=ths[x-1+(y-1)*w32];}
        if(y<h32-1) {num++; sum+=ths[x-1+(y+1)*w32];}
        num++; sum+=ths[x-1+(y)*w32];
    }
    if(x<w32-1)
    {
        if(y>0) {num++; sum+=ths[x+1+(y-1)*w32];}
        if(y<h32-1) {num++; sum+=ths[x+1+(y+1)*w32];}
        num++; sum+=ths[x+1+(y)*w32];
    }
    if(y>0) {num++; sum+=ths[x+(y-1)*w32];}
    if(y<h32-1) {num++; sum+=ths[x+(y+1)*w32];}
    num++; sum+=ths[x+y*w32];
    thsSmoothed[x+y*w32] = (sum/num) * (sum/num);
}
```

Gradient Histogram 방법을 통해 이미지를 32x32 영역으로 나눈 후 각 영역마다 히스토그램을 생성하고 threshold + smoothed threshold를 계산하는 함수

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/PixelSelector2.cpp#L78>

5. Pixel Selection

CODE REVIEW

PixelSelector2.cpp::makeHists()::L78

```
float * mapmax0 = fh->absSquaredGrad[0];
int w = WG[0];
int h = hG[0];
int w32 = w/32;
int h32 = h/32;
thsStep = w32;
for(int y=0;y<h32;y++)
for(int x=0;x<w32;x++) {
    float* map0 = mapmax0+32*x+32*y*w;
    int* hist0 = gradHist;// + 50*(x+y*w32);
    memset(hist0,0,sizeof(int)*50);
    for(int j=0;j<32;j++) for(int i=0;i<32;i++) {
        int it = i+32*x;
        int jt = j+32*y;
        if(it>w-2 || jt>h-2 || it<1 || jt<1) continue;
        int g = sqrtf(map0[i+j*w]);
        if(g>48) g=48;
        hist0[g+1]++;
        hist0[0]++;
    }
    ths[x+y*w32] = computeHistQuantil(hist0,setting_minGradHistCut) + setting_minGradHistAdd;
}
for(int y=0;y<h32;y++)
for(int x=0;x<w32;x++) {
    float sum=0,num=0;
    if(x>0)
    {
        if(y>0) {num++; sum+=ths[x-1+(y-1)*w32];}
        if(y<h32-1) {num++; sum+=ths[x-1+(y+1)*w32];}
        num++; sum+=ths[x-1+(y)*w32];
    }
    if(x<w32-1)
    {
        if(y>0) {num++; sum+=ths[x+1+(y-1)*w32];}
        if(y<h32-1) {num++; sum+=ths[x+1+(y+1)*w32];}
        num++; sum+=ths[x+1+(y)*w32];
    }
    if(y>0) {num++; sum+=ths[x+(y-1)*w32];}
    if(y<h32-1) {num++; sum+=ths[x+(y+1)*w32];}
    num++; sum+=ths[x+y*w32];
    thsSmoothed[x+y*w32] = (sum/num) * (sum/num);
}
```

Level0(원본)에서 각 픽셀별로 Absolute Squared Gradient 값을 mapmax0 포인터로 가리킨다.

$$\text{absSquaredGrad}[0][i] = dx_i^2 + dy_i^2$$

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/PixelSelector2.cpp#L78>

5. Pixel Selection

CODE REVIEW

PixelSelector2.cpp::makeHists()::L78

```
float * mapmax0 = fh->absSquaredGrad[0];
int w = wG[0];
int h = hG[0];
int w32 = w/32;
int h32 = h/32;
thsStep = w32;
for(int y=0;y<h32;y++) {
    for(int x=0;x<w32;x++) {
        float* map0 = mapmax0+32*x+32*y*w;
        int* hist0 = gradHist; // + 50*(x+y*w32);
        memset(hist0,0,sizeof(int)*50);
        for(int j=0;j<32;j++) for(int i=0;i<32;i++){
            int it = i+32*x;
            int jt = j+32*y;
            if(it>w-2 || jt>h-2 || it<1 || jt<1) continue;
            int g = sqrtf(map0[i+j*w]);
            if(g>48) g=48;
            hist0[g+1]++;
            hist0[0]++;
        }
        ths[x+y*w32] = computeHistQuantil(hist0,setting_minGradHistCut) + setting_minGradHistAdd;
    }
    for(int y=0;y<h32;y++) {
        for(int x=0;x<w32;x++) {
            float sum=0,num=0;
            if(x>0)
            {
                if(y>0) {num++; sum+=ths[x-1+(y-1)*w32];}
                if(y<h32-1) {num++; sum+=ths[x-1+(y+1)*w32];}
                num++; sum+=ths[x-1+(y)*w32];
            }
            if(x<w32-1)
            {
                if(y>0) {num++; sum+=ths[x+1+(y-1)*w32];}
                if(y<h32-1) {num++; sum+=ths[x+1+(y+1)*w32];}
                num++; sum+=ths[x+1+(y)*w32];
            }
            if(y>0) {num++; sum+=ths[x+(y-1)*w32];}
            if(y<h32-1) {num++; sum+=ths[x+(y+1)*w32];}
            num++; sum+=ths[x+y*w32];
            thsSmoothed[x+y*w32] = (sum/num) * (sum/num);
        }
    }
}
```

Level0(원본)에서 이미지의 height, width를 변수에 저장하고 이를 32등분한 값도 변수에 저장한다.
한 번의 스텝 당 w32 값씩 이동한다.

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/PixelSelector2.cpp#L78>

5. Pixel Selection

CODE REVIEW

PixelSelector2.cpp::makeHists()::L78

```
float * mapmax0 = fh->absSquaredGrad[0];
int w = wG[0];
int h = hG[0];
int w32 = w/32;
int h32 = h/32;
thsStep = w32;
for(int y=0;y<h32;y++)
for(int x=0;x<w32;x++) {
    float* map0 = mapmax0+32*x+32*y*w;
    int* hist0 = gradHist;// + 50*(x+y*w32);
    memset(hist0,0,sizeof(int)*50);
    for(int j=0;j<32;j++) for(int i=0;i<32;i++){
        int it = i+32*x;
        int jt = j+32*y;
        if(it>w-2 || jt>h-2 || it<1 || jt<1) continue;
        int g = sqrtf(map0[i+j]*w);
        if(g>48) g=48;
        hist0[g+1]++;
        hist0[0]++;
    }
    ths[x+y*w32] = computeHistQuantil(hist0,setting_minGradHistCut) + setting_minGradHistAdd;
}
for(int y=0;y<h32;y++)
for(int x=0;x<w32;x++) {
    float sum=0,num=0;
    if(x>0)
    {
        if(y>0) {num++; sum+=ths[x-1+(y-1)*w32];}
        if(y<h32-1) {num++; sum+=ths[x-1+(y+1)*w32];}
        num++; sum+=ths[x-1+(y)*w32];
    }
    if(x<w32-1)
    {
        if(y>0) {num++; sum+=ths[x+1+(y-1)*w32];}
        if(y<h32-1) {num++; sum+=ths[x+1+(y+1)*w32];}
        num++; sum+=ths[x+1+(y)*w32];
    }
    if(y>0) {num++; sum+=ths[x+(y-1)*w32];}
    if(y<h32-1) {num++; sum+=ths[x+(y+1)*w32];}
    num++; sum+=ths[x+y*w32];
    thsSmoothed[x+y*w32] = (sum/num) * (sum/num);
}
```

한 영역 내에 있는 모든 픽셀의 루프를 돌면서 $g = \sqrt{dx_i^2 + dy_i^2}$ 값을 계산하고 이를 히스토그램에 카운팅한다.
g값은 이론적으로 0~360의 값을 가질 수 있는데 50이 넘는 경우 50으로 카운팅한다.
Hist0[0] 값에는 모든 카운트의 총합이 저장된다.

360 $\sim \sqrt{255^2 + 255^2}$

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/PixelSelector2.cpp#L78>

5. Pixel Selection

CODE REVIEW

PixelSelector2.cpp::makeHists()::L78

```
float * mapmax0 = fh->absSquaredGrad[0];
int w = wG[0];
int h = hG[0];
int w32 = w/32;
int h32 = h/32;
thsStep = w32;
for(int y=0;y<h32;y++) {
    for(int x=0;x<w32;x++) {
        float* map0 = mapmax0+32*x+32*y*w;
        int* hist0 = gradHist; // + 50*(x+y*w32);
        memset(hist0,0,sizeof(int)*50);
        for(int j=0;j<32;j++) for(int i=0;i<32;i++){
            int it = i+32*x;
            int jt = j+32*y;
            if(it>w-2 || jt>h-2 || it<1 || jt<1) continue;
            int g = sqrtf(map0[i+j*w]);
            if(g>48) g=48;
            hist0[g+1]++;
            hist0[0]++;
        }
        ths[x+y*w32] = computeHistQuantil(hist0,setting_minGradHistCut) + setting_minGradHistAdd;
    }
    for(int y=0;y<h32;y++) {
        for(int x=0;x<w32;x++) {
            float sum=0,num=0;
            if(x>0)
            {
                if(y>0) {num++; sum+=ths[x-1+(y-1)*w32];}
                if(y<h32-1) {num++; sum+=ths[x-1+(y+1)*w32];}
                num++; sum+=ths[x-1+(y)*w32];
            }
            if(x<w32-1)
            {
                if(y>0) {num++; sum+=ths[x+1+(y-1)*w32];}
                if(y<h32-1) {num++; sum+=ths[x+1+(y+1)*w32];}
                num++; sum+=ths[x+1+(y)*w32];
            }
            if(y>0) {num++; sum+=ths[x+(y-1)*w32];}
            if(y<h32-1) {num++; sum+=ths[x+(y+1)*w32];}
            num++; sum+=ths[x+y*w32];
            thsSmoothed[x+y*w32] = (sum/num) * (sum/num);
        }
    }
}
```

히스토그램에서 총합의 50% 이상이 되는 bin을 threshold로 설정한다.

이 때 setting_minGradHistAdd (=7) 값을 최소값으로 넣어준다.

$8 \leq \text{ths} \leq 50$

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/PixelSelector2.cpp#L78>

5. Pixel Selection

CODE REVIEW

PixelSelector2.cpp::makeHists()::L78

```
float * mapmax0 = fh->absSquaredGrad[0];
int w = wG[0];
int h = hG[0];
int w32 = w/32;
int h32 = h/32;
thsStep = w32;
for(int y=0;y<h32;y++) {
    for(int x=0;x<w32;x++) {
        float* map0 = mapmax0+32*x+32*y*w;
        int* hist0 = gradHist; // + 50*(x+y*w32);
        memset(hist0,0,sizeof(int)*50);
        for(int j=0;j<32;j++) for(int i=0;i<32;i++){
            int it = i+32*x;
            int jt = j+32*y;
            if(it>w-2 || jt>h-2 || it<1 || jt<1) continue;
            int g = sqrtf(map0[i+j*w]);
            if(g>48) g=48;
            hist0[g+1]++;
            hist0[0]++;
        }
        ths[x+y*w32] = computeHistQuantil(hist0,setting_minGradHistCut) + setting_minGradHistAdd;
    }
    for(int y=0;y<h32;y++) {
        for(int x=0;x<w32;x++) {
            float sum=0,num=0;
            if(x>0)
            {
                if(y>0) {num++; sum+=ths[x-1+(y-1)*w32];}
                if(y<h32-1) {num++; sum+=ths[x-1+(y+1)*w32];}
                num++; sum+=ths[x-1+(y)*w32];
            }
            if(x<w32-1)
            {
                if(y>0) {num++; sum+=ths[x+1+(y-1)*w32];}
                if(y<h32-1) {num++; sum+=ths[x+1+(y+1)*w32];}
                num++; sum+=ths[x+1+(y)*w32];
            }
            if(y>0) {num++; sum+=ths[x+(y-1)*w32];}
            if(y<h32-1) {num++; sum+=ths[x+(y+1)*w32];}
            num++; sum+=ths[x+y*w32];
            thsSmoothed[x+y*w32] = (sum/num) * (sum/num);
        }
    }
}
```

32x32 Grid를 다시 3x3 Grid씩 묶은 다음 threshold의 평균을 구한다.

이를 통해 조금 더 완곡한 (Smoothed) threshold 값을 계산할 수 있다.

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/PixelSelector2.cpp#L78>

5. Pixel Selection

CODE REVIEW

PixelSelector2.cpp::select():L295

```
Eigen::Vector3i PixelSelector::select(const FrameHessian*
const fh, float* map_out, int pot, float thFactor) {
    Eigen::Vector3f const * const map0 = fh->dI;

    float * mapmax0 = fh->absSquaredGrad[0];
    float * mapmax1 = fh->absSquaredGrad[1];
    float * mapmax2 = fh->absSquaredGrad[2];
    int w = wG[0];
    int w1 = wG[1];
    int w2 = wG[2];
    int h = hG[0];

    const Vec2f directions[16] = {
        Vec2f(0, 1.0000),
        Vec2f(0.3827, 0.9239),
        Vec2f(0.1951, 0.9808),
        Vec2f(0.9239, 0.3827),
        Vec2f(0.7071, 0.7071),
        Vec2f(0.3827, -0.9239),
        Vec2f(0.8315, 0.5556),
        Vec2f(0.8315, -0.5556),
        Vec2f(0.5556, -0.8315),
        Vec2f(0.9808, 0.1951),
        Vec2f(0.9239, -0.3827),
        Vec2f(0.7071, -0.7071),
        Vec2f(0.5556, 0.8315),
        Vec2f(0.9808, -0.1951),
        Vec2f(1.0000, 0.0000),
        Vec2f(0.1951, -0.9808);

    memset(map_out, 0, w*h*sizeof(PixelSelectorStatus));
    float dw1 = setting_gradDownweightPerLevel;
    float dw2 = dw1*dw1;
```

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/PixelSelector2.cpp#L295>

5. Pixel Selection

CODE REVIEW

PixelSelector2.cpp::select():L295

```
Eigen::Vector3i PixelSelector::select(const FrameHessian*  
const fh, float* map_out, int pot, float thFactor) {  
    Eigen::Vector3f const * const map0 = fh->dI;  
  
    float * mapmax0 = fh->absSquaredGrad[0];  
    float * mapmax1 = fh->absSquaredGrad[1];  
    float * mapmax2 = fh->absSquaredGrad[2];  
    int w = wG[0];  
    int w1 = wG[1];  
    int w2 = wG[2];  
    int h = hG[0];  
  
    const Vec2f directions[16] = {  
        Vec2f(0, 1.0000),  
        Vec2f(0.3827, 0.9239),  
        Vec2f(0.1951, 0.9808),  
        Vec2f(0.9239, 0.3827),  
        Vec2f(0.7071, 0.7071),  
        Vec2f(0.3827, -0.9239),  
        Vec2f(0.8315, 0.5556),  
        Vec2f(0.8315, -0.5556),  
        Vec2f(0.5556, -0.8315),  
        Vec2f(0.9808, 0.1951),  
        Vec2f(0.9239, -0.3827),  
        Vec2f(0.7071, -0.7071),  
        Vec2f(0.5556, 0.8315),  
        Vec2f(0.9808, -0.1951),  
        Vec2f(1.0000, 0.0000),  
        Vec2f(0.1951, -0.9808);  
  
    memset(map_out, 0, w*h*sizeof(PixelSelectorStatus));  
    float dw1 = setting_gradDownweightPerLevel;  
    float dw2 = dw1*dw1;
```

Dynamic Grid 방법을 사용하여 Pixel map을 생성하는 함수.

이미지에서 픽셀을 선택하여 Pixel map에 저장하므로 select()로 작명한 듯 하다.

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/PixelSelector2.cpp#L295>

5. Pixel Selection

CODE REVIEW

PixelSelector2.cpp::select():L295

```
Eigen::Vector3i PixelSelector::select(const FrameHessian*  
const fh, float* map_out, int pot, float thFactor) {  
    Eigen::Vector3f const * const map0 = fh->di;  
  
    float * mapmax0 = fh->absSquaredGrad[0];  
    float * mapmax1 = fh->absSquaredGrad[1];  
    float * mapmax2 = fh->absSquaredGrad[2];  
    int w = wG[0];  
    int w1 = wG[1];  
    int w2 = wG[2];  
    int h = hG[0];  
  
    const Vec2f directions[16] = {  
        Vec2f(0, 1.0000),  
        Vec2f(0.3827, 0.9239),  
        Vec2f(0.1951, 0.9808),  
        Vec2f(0.9239, 0.3827),  
        Vec2f(0.7071, 0.7071),  
        Vec2f(0.3827, -0.9239),  
        Vec2f(0.8315, 0.5556),  
        Vec2f(0.8315, -0.5556),  
        Vec2f(0.5556, -0.8315),  
        Vec2f(0.9808, 0.1951),  
        Vec2f(0.9239, -0.3827),  
        Vec2f(0.7071, -0.7071),  
        Vec2f(0.5556, 0.8315),  
        Vec2f(0.9808, -0.1951),  
        Vec2f(1.0000, 0.0000),  
        Vec2f(0.1951, -0.9808);  
  
    memset(map_out, 0, w*h*sizeof(PixelSelectorStatus));  
    float dw1 = setting_gradDownweightPerLevel;  
    float dw2 = dw1*dw1;
```

현재 프레임에서 di 값을 가져온다. di에는 레벨0 이미지에 대한 정보가 들어있다.

di[0] = pixel intensity
di[1] = dx
di[2] = dy

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/PixelSelector2.cpp#L295>

5. Pixel Selection

CODE REVIEW

PixelSelector2.cpp::select():L295

```
Eigen::Vector3i PixelSelector::select(const FrameHessian*
const fh, float* map_out, int pot, float thFactor) {
    Eigen::Vector3f const * const map0 = fh->dI;

    float * mapmax0 = fh->absSquaredGrad[0];
    float * mapmax1 = fh->absSquaredGrad[1];
    float * mapmax2 = fh->absSquaredGrad[2];
    int w = WG[0];
    int w1 = WG[1];
    int w2 = WG[2];
    int h = hG[0];

    const Vec2f directions[16] = {
        Vec2f(0, 1.0000),
        Vec2f(0.3827, 0.9239),
        Vec2f(0.1951, 0.9808),
        Vec2f(0.9239, 0.3827),
        Vec2f(0.7071, 0.7071),
        Vec2f(0.3827, -0.9239),
        Vec2f(0.8315, 0.5556),
        Vec2f(0.8315, -0.5556),
        Vec2f(0.5556, -0.8315),
        Vec2f(0.9808, 0.1951),
        Vec2f(0.9239, -0.3827),
        Vec2f(0.7071, -0.7071),
        Vec2f(0.5556, 0.8315),
        Vec2f(0.9808, -0.1951),
        Vec2f(1.0000, 0.0000),
        Vec2f(0.1951, -0.9808);

    memset(map_out, 0, w*h*sizeof(PixelSelectorStatus));
    float dw1 = setting_gradDownweightPerLevel;
    float dw2 = dw1*dw1;
```

피라미드 레벨 0,1,2에서 이미지의 Absolute Squared Gradient 값을 가져온다.

absSquaredGrad에는 각 픽셀별로 아래의 값이 저장되어 있다.

$$\text{absSquaredGrad}[level][i] = dx_i^2 + dy_i^2$$

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/PixelSelector2.cpp#L295>

5. Pixel Selection

CODE REVIEW

PixelSelector2.cpp::select():L295

```
Eigen::Vector3i PixelSelector::select(const FrameHessian*
const fh, float* map_out, int pot, float thFactor) {
    Eigen::Vector3f const * const map0 = fh->dI;

    float * mapmax0 = fh->absSquaredGrad[0];
    float * mapmax1 = fh->absSquaredGrad[1];
    float * mapmax2 = fh->absSquaredGrad[2];
    int w = WG[0];
    int w1 = WG[1];
    int w2 = WG[2];
    int h = hG[0];
```

피라미드 레벨 1,2에서 width 값과
레벨0에서 width, height 값을 저장한다.

```
    const Vec2f directions[16] = {
        Vec2f(0, 1.0000),
        Vec2f(0.3827, 0.9239),
        Vec2f(0.1951, 0.9808),
        Vec2f(0.9239, 0.3827),
        Vec2f(0.7071, 0.7071),
        Vec2f(0.3827, -0.9239),
        Vec2f(0.8315, 0.5556),
        Vec2f(0.8315, -0.5556),
        Vec2f(0.5556, -0.8315),
        Vec2f(0.9808, 0.1951),
        Vec2f(0.9239, -0.3827),
        Vec2f(0.7071, -0.7071),
        Vec2f(0.5556, 0.8315),
        Vec2f(0.9808, -0.1951),
        Vec2f(1.0000, 0.0000),
        Vec2f(0.1951, -0.9808);

        memset(map_out, 0, w*h*sizeof(PixelSelectorStatus));
        float dw1 = setting_gradDownweightPerLevel;
        float dw2 = dw1*dw1;
```

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/PixelSelector2.cpp#L295>

5. Pixel Selection

CODE REVIEW

PixelSelector2.cpp::select():L295

```
Eigen::Vector3i PixelSelector::select(const FrameHessian*
const fh, float* map_out, int pot, float thFactor) {
    Eigen::Vector3f const * const map0 = fh->dI;

    float * mapmax0 = fh->absSquaredGrad[0];
    float * mapmax1 = fh->absSquaredGrad[1];
    float * mapmax2 = fh->absSquaredGrad[2];
    int w = wG[0];
    int w1 = wG[1];
    int w2 = wG[2];
    int h = hG[0];

    const Vec2f directions[16] = {
        Vec2f(0, 1.0000),
        Vec2f(0.3827, 0.9239),
        Vec2f(0.1951, 0.9808),
        Vec2f(0.9239, 0.3827),
        Vec2f(0.7071, 0.7071),
        Vec2f(0.3827, -0.9239),
        Vec2f(0.8315, 0.5556),
        Vec2f(0.8315, -0.5556),
        Vec2f(0.5556, -0.8315),
        Vec2f(0.9808, 0.1951),
        Vec2f(0.9239, -0.3827),
        Vec2f(0.7071, -0.7071),
        Vec2f(0.5556, 0.8315),
        Vec2f(0.9808, -0.1951),
        Vec2f(1.0000, 0.0000),
        Vec2f(0.1951, -0.9808)};
    memset(map_out, 0, w*h*sizeof(PixelSelectorStatus));
    float dw1 = setting_gradDownweightPerLevel;
    float dw2 = dw1*dw1;
```

픽셀 추출이 특정 영역에 쏠리는 것을 방지하기 위해 랜덤한 방향벡터를 생성한다.

randdir

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/PixelSelector2.cpp#L295>

5. Pixel Selection

CODE REVIEW

PixelSelector2.cpp::select():L295

```
Eigen::Vector3i PixelSelector::select(const FrameHessian*
const fh, float* map_out, int pot, float thFactor) {
    Eigen::Vector3f const * const map0 = fh->dI;

    Float * mapmax0 = fh->absSquaredGrad[0];
    float * mapmax1 = fh->absSquaredGrad[1];
    float * mapmax2 = fh->absSquaredGrad[2];
    int w = wG[0];
    int w1 = wG[1];
    int w2 = wG[2];
    int h = hG[0];

    const Vec2f directions[16] = {
        Vec2f(0, 1.0000),
        Vec2f(0.3827, 0.9239),
        Vec2f(0.1951, 0.9808),
        Vec2f(0.9239, 0.3827),
        Vec2f(0.7071, 0.7071),
        Vec2f(0.3827, -0.9239),
        Vec2f(0.8315, 0.5556),
        Vec2f(0.8315, -0.5556),
        Vec2f(0.5556, -0.8315),
        Vec2f(0.9808, 0.1951),
        Vec2f(0.9239, -0.3827),
        Vec2f(0.7071, -0.7071),
        Vec2f(0.5556, 0.8315),
        Vec2f(0.9808, -0.1951),
        Vec2f(1.0000, 0.0000),
        Vec2f(0.1951, -0.9808)};

    memset(map_out, 0, w*h*sizeof(PixelSelectorStatus));
    float dw1 = setting_gradDownweightPerLevel;
    float dw2 = dw1*dw1;
```

map_out 포인터에 최종적으로 선택된 픽셀의 타입 (1,2,4)를 저장하기 위해 초기화한다.

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/PixelSelector2.cpp#L295>

5. Pixel Selection

CODE REVIEW

PixelSelector2.cpp::select():L295

```
Eigen::Vector3i PixelSelector::select(const FrameHessian*
const fh, float* map_out, int pot, float thFactor) {
    Eigen::Vector3f const * const map0 = fh->dI;

    Float * mapmax0 = fh->absSquaredGrad[0];
    float * mapmax1 = fh->absSquaredGrad[1];
    float * mapmax2 = fh->absSquaredGrad[2];
    int w = wG[0];
    int w1 = wG[1];
    int w2 = wG[2];
    int h = hG[0];

    const Vec2f directions[16] = {
        Vec2f(0, 1.0000),
        Vec2f(0.3827, 0.9239),
        Vec2f(0.1951, 0.9808),
        Vec2f(0.9239, 0.3827),
        Vec2f(0.7071, 0.7071),
        Vec2f(0.3827, -0.9239),
        Vec2f(0.8315, 0.5556),
        Vec2f(0.8315, -0.5556),
        Vec2f(0.5556, -0.8315),
        Vec2f(0.9808, 0.1951),
        Vec2f(0.9239, -0.3827),
        Vec2f(0.7071, -0.7071),
        Vec2f(0.5556, 0.8315),
        Vec2f(0.9808, -0.1951),
        Vec2f(1.0000, 0.0000),
        Vec2f(0.1951, -0.9808);

    memset(map_out, 0, w*h*sizeof(PixelSelectorStatus));
    float dw1 = setting_gradDownweightPerLevel;
    float dw2 = dw1*dw1;
```

피라미드 레벨이 높아질수록 down weight (=0.75) 값을 곱해서

제일 낮은 레벨(Lv0, 원본)에서 많은 픽셀이 추출될 수 있도록 유도한다.

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/PixelSelector2.cpp#L295>

5. Pixel Selection

CODE REVIEW

PixelSelector2.cpp::select():L295

```
float pixelTH0 = thsSmoothed[(xf>>5) + (yf>>5) * thsStep];
float pixelTH1 = pixelTH0*dw1;
float pixelTH2 = pixelTH1*dw2;
float ag0 = mapmax0[idx];

if(ag0 > pixelTH0*thFactor)
{
    Vec2f ag0d = map0[idx].tail<2>();
    float dirNorm = fabsf((float)(ag0d.dot(dir2)));
    if(!setting_selectDirectionDistribution) dirNorm = ag0;
    if(dirNorm > bestVal2)
        { bestVal2 = dirNorm; bestIdx2 = idx; bestIdx3 = -2; bestIdx4 = -2; }
}

if(bestIdx3===-2) continue;

float ag1 = mapmax1[(int)(xf*0.5f+0.25f) + (int)(yf*0.5f+0.25f)*w1];

if(ag1 > pixelTH1*thFactor)
{
    Vec2f ag0d = map0[idx].tail<2>();
    float dirNorm = fabsf((float)(ag0d.dot(dir3)));
    if(!setting_selectDirectionDistribution) dirNorm = ag1;
    if(dirNorm > bestVal3)
        { bestVal3 = dirNorm; bestIdx3 = idx; bestIdx4 = -2; }
}

if(bestIdx4===-2) continue;

float ag2 = mapmax2[(int)(xf*0.25f+0.125) + (int)(yf*0.25f+0.125)*w2];

if(ag2 > pixelTH2*thFactor)
{
    Vec2f ag0d = map0[idx].tail<2>();
    float dirNorm = fabsf((float)(ag0d.dot(dir4)));
    if(!setting_selectDirectionDistribution) dirNorm = ag2;
    if(dirNorm > bestVal4)
        { bestVal4 = dirNorm; bestIdx4 = idx; }
}
```

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/PixelSelector2.cpp#L295>

5. Pixel Selection

CODE REVIEW

PixelSelector2.cpp::select():L295

```
Float pixelTH0 = thsSmoothed[(xf>>5) + (yf>>5) * thsStep];
Float pixelTH1 = pixelTH0*dw1;
Float pixelTH2 = pixelTH1*dw2;
Float ag0 = mapmax0[idx];

if(ag0 > pixelTH0*thFactor)
{
    Vec2f ag0d = map0[idx].tail<2>();
    float dirNorm = fabsf((float)(ag0d.dot(dir2)));
    if(!setting_selectDirectionDistribution) dirNorm = ag0;
    if(dirNorm > bestVal2)
    { bestVal2 = dirNorm; bestIdx2 = idx; bestIdx3 = -2; bestIdx4 = -2; }
}

if(bestIdx3 == -2) continue;

float ag1 = mapmax1[(int)(xf*0.5f+0.25f) + (int)(yf*0.5f+0.25f)*w1];

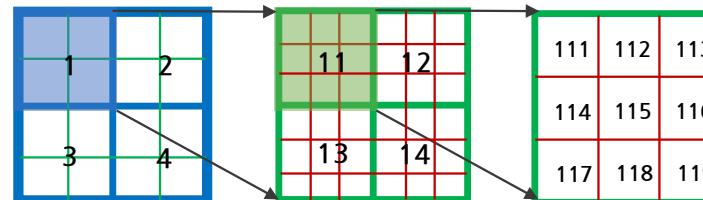
if(ag1 > pixelTH1*thFactor)
{
    Vec2f ag0d = map0[idx].tail<2>();
    float dirNorm = fabsf((float)(ag0d.dot(dir3)));
    if(!setting_selectDirectionDistribution) dirNorm = ag1;
    if(dirNorm > bestVal3)
    { bestVal3 = dirNorm; bestIdx3 = idx; bestIdx4 = -2; }
}

if(bestIdx4 == -2) continue;

float ag2 = mapmax2[(int)(xf*0.25f+0.125) + (int)(yf*0.25f+0.125)*w2];

if(ag2 > pixelTH2*thFactor)
{
    Vec2f ag0d = map0[idx].tail<2>();
    float dirNorm = fabsf((float)(ag0d.dot(dir4)));
    if(!setting_selectDirectionDistribution) dirNorm = ag2;
    if(dirNorm > bestVal4)
    { bestVal4 = dirNorm; bestIdx4 = idx; }
}
```

8중 for문을 돌면서 레벨0에서부터 픽셀을 추출한다.



<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/PixelSelector2.cpp#L295>

5. Pixel Selection

레벨0,1,2 별로 레벨이 높아질수록 threshold 값을 0.75배씩 감소시키면서 추출될 확률을 높인다.

현재 픽셀의 Absolute Squared Gradient 값을 구한다. ($ag0 = dx^2 + dy^2$)

PixelSelector2.cpp::select():L295

```

Float pixelTH0 = thsSmoothed[(xf>>5) + (yf>>5) * thsStep];
Float pixelTH1 = pixelTH0*dw1;
Float pixelTH2 = pixelTH1*dw2;
Float ag0 = mapmax0[idx];

if(ag0 > pixelTH0*thFactor)
{
    Vec2f ag0d = map0[idx].tail<2>();
    float dirNorm = fabsf((float)(ag0d.dot(dir2)));
    if(!setting_selectDirectionDistribution) dirNorm = ag0;
    if(dirNorm > bestVal2)
        { bestVal2 = dirNorm; bestIdx2 = idx; bestIdx3 = -2; bestIdx4 = -2; }
}

if(bestIdx3 == -2) continue;

float ag1 = mapmax1[(int)(xf*0.5f+0.25f) + (int)(yf*0.5f+0.25f)*w1];

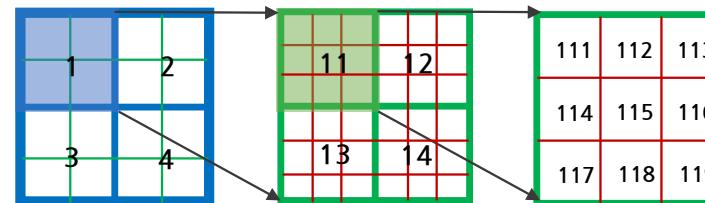
if(ag1 > pixelTH1*thFactor)
{
    Vec2f ag0d = map0[idx].tail<2>();
    float dirNorm = fabsf((float)(ag0d.dot(dir3)));
    if(!setting_selectDirectionDistribution) dirNorm = ag1;
    if(dirNorm > bestVal3)
        { bestVal3 = dirNorm; bestIdx3 = idx; bestIdx4 = -2; }

if(bestIdx4 == -2) continue;

float ag2 = mapmax2[(int)(xf*0.25f+0.125) + (int)(yf*0.25f+0.125)*w2];

if(ag2 > pixelTH2*thFactor)
{
    Vec2f ag0d = map0[idx].tail<2>();
    float dirNorm = fabsf((float)(ag0d.dot(dir4)));
    if(!setting_selectDirectionDistribution) dirNorm = ag2;
    if(dirNorm > bestVal4)
        { bestVal4 = dirNorm; bestIdx4 = idx; }
}

```



5. Pixel Selection

CODE REVIEW

PixelSelector2.cpp::select():L295

```

float pixelTH0 = thsSmoothed[(xf>>5) + (yf>>5) * thsStep];
float pixelTH1 = pixelTH0*dw1;
float pixelTH2 = pixelTH1*dw2;
float ag0 = mapmax0[idx];
map0[idx].tail<2>() = (dx,dy)

if(ag0 > pixelTH0*thFactor)
{
    Vec2f ag0d = map0[idx].tail<2>();
    float dirNorm = fabsf((float)(ag0d.dot(dir2)));
    if(!setting_selectDirectionDistribution) dirNorm = ag0;
    if(dirNorm > bestVal2)
        { bestVal2 = dirNorm; bestIdx2 = idx; bestIdx3 = -2; bestIdx4 = -2; }

    if(bestIdx3 == -2) continue;

    float ag1 = mapmax1[(int)(xf*0.5f+0.25f) + (int)(yf*0.5f+0.25f)*w1];

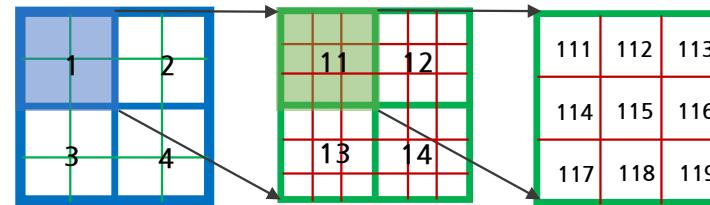
    if(ag1 > pixelTH1*thFactor)
    {
        Vec2f ag0d = map0[idx].tail<2>();
        float dirNorm = fabsf((float)(ag0d.dot(dir3)));
        if(!setting_selectDirectionDistribution) dirNorm = ag1;
        if(dirNorm > bestVal3)
            { bestVal3 = dirNorm; bestIdx3 = idx; bestIdx4 = -2; }

        if(bestIdx4 == -2) continue;

        float ag2 = mapmax2[(int)(xf*0.25f+0.125) + (int)(yf*0.25f+0.125)*w2];

        if(ag2 > pixelTH2*thFactor)
        {
            Vec2f ag0d = map0[idx].tail<2>();
            float dirNorm = fabsf((float)(ag0d.dot(dir4)));
            if(!setting_selectDirectionDistribution) dirNorm = ag2;
            if(dirNorm > bestVal4)
                { bestVal4 = dirNorm; bestIdx4 = idx; }
        }
    }
}

```



레벨0(원본) 픽셀부터 $dx^2 + dy^2 > \text{thsSmoothed}_{\text{level}0}$ 값을 비교하고 만약 해당 조건을 만족한다면 해당 픽셀의 (dx,dy)와 랜덤한 방향벡터 dirX의 norm을 구한다.

$$\text{randdir} \cdot (dx, dy) > 0$$

그리고 위의 조건 또한 만족한다면 다른 레벨은 고려하지 않고 해당 픽셀을 를 대표하는 픽셀로 간주하고 루프를 탈출한다.

5. Pixel Selection

CODE REVIEW

PixelSelector2.cpp::select():L295

```
float pixelTH0 = thsSmoothed[(xf>>5) + (yf>>5) * thsStep];
float pixelTH1 = pixelTH0*dw1;
float pixelTH2 = pixelTH1*dw2;
float ag0 = mapmax0[idx];

if(ag0 > pixelTH0*thFactor)
{
    Vec2f ag0d = map0[idx].tail<2>();
    float dirNorm = fabsf((float)(ag0d.dot(dir2)));
    if(!setting_selectDirectionDistribution) dirNorm = ag0;
    if(dirNorm > bestVal2)
        { bestVal2 = dirNorm; bestIdx2 = idx; bestIdx3 = -2; bestIdx4 = -2; }

if(bestIdx3 == -2) continue;

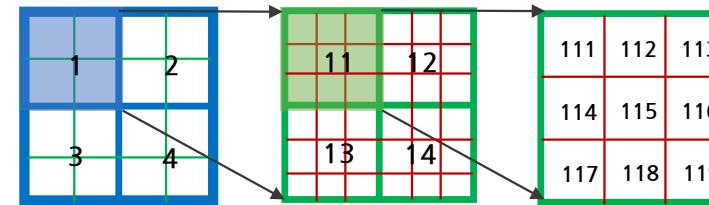
float ag1 = mapmax1[(int)(xf*0.5f+0.25f) + (int)(yf*0.5f+0.25f)*w1];

if(ag1 > pixelTH1*thFactor)
{
    Vec2f ag0d = map0[idx].tail<2>();
    float dirNorm = fabsf((float)(ag0d.dot(dir3)));
    if(!setting_selectDirectionDistribution) dirNorm = ag1;
    if(dirNorm > bestVal3)
        { bestVal3 = dirNorm; bestIdx3 = idx; bestIdx4 = -2; }

if(bestIdx4 == -2) continue;

float ag2 = mapmax2[(int)(xf*0.25f+0.125) + (int)(yf*0.25f+0.125)*w2];

if(ag2 > pixelTH2*thFactor)
{
    Vec2f ag0d = map0[idx].tail<2>();
    float dirNorm = fabsf((float)(ag0d.dot(dir4)));
    if(!setting_selectDirectionDistribution) dirNorm = ag2;
    if(dirNorm > bestVal4)
        { bestVal4 = dirNorm; bestIdx4 = idx; }
}
```



레벨0에서 조건을 만족하지 못한 경우 레벨1,2로 점차 올라가면서 검사한다.

5. Pixel Selection

CODE REVIEW

PixelSelector2.cpp::select():L295

```
if(bestIdx2>0)
{
    map_out[bestIdx2] = 1;
    bestVal3 = 1e10;
    n2++;
}
}
if(bestIdx3>0)
{
    map_out[bestIdx3] = 2;
    bestVal4 = 1e10;
    n3++;
}
}
if(bestIdx4>0)
{
    map_out[bestIdx4] = 4;
    n4++;
}
```

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/PixelSelector2.cpp#L295>

5. Pixel Selection

CODE REVIEW

PixelSelector2.cpp::select():L295

```
if(bestIdx2>0)
{
    map_out[bestIdx2] = 1;
    bestVal3 = 1e10;
    n2++;
}

if(bestIdx3>0)
{
    map_out[bestIdx3] = 2;
    bestVal4 = 1e10;
    n3++;
}

if(bestIdx4>0)
{
    map_out[bestIdx4] = 4;
    n4++;
}
```

Pixel map에는 Lv0에서 추출된 포인트일 경우 1, Lv1에서 추출된 포인트는 2,
Lv3에서 추출된 포인트는 4의 값이 해당 픽셀 위치에 저장된다. 포인트가 추출되지
않은 영역의 값은 저장되지 않는다. (≈ 0)

map_out

1	2	4			2						
											1
		4			2						
			1								
1						2					
							4				
			1		2						
								4			

모든 픽셀을 그리지 않고 12x12 Grid만 그린 그림

글씨가 Grid 영역보다 작은 이유는 글씨가 한 픽셀을 의미하기 때문

<https://github.com/JakobEngel/dso/blob/master/src/FullSystem/PixelSelector2.cpp#L295>